AFRL-RY-WP-TR-2011-1310

TEST AND EVALUATION OF ARCHITECTURE-AWARE COMPILER ENVIRONMENT

Allan Snavely

University of California, San Diego

NOVEMBER 2011
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency's Public Release Center and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2011-1310 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//signed//

KERRY HILL, Program Manager
Advanced Sensor Components Branch
Aerospace Components Division

//signed//

BRADLEY J. PAUL, Chief
Advanced Sensor Components Branch
Aerospace Components Division

//signed//

BRADLEY CHRISTIAN, Lt Col, USAF
Deputy Division Chief
Aerospace Components Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

1. REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
November 2011	Final	26 February 2009 –31 November 2011

4. TITLE AND SUBTITLE
TEST AND EVALUATION OF ARCHITECTURE-AWARE COMPILER ENVIRONMENT

5a. CONTRACT NUMBER
FA8650-09-C-7917

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Allan Snavely

5d. PROJECT NUMBER
ARPR

5e. TASK NUMBER
YD

5f. WORK UNIT NUMBER
ARPRYD1E

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

UCSD,
Office of Contract & Grant Admin
9500 Gilman Dr, Dept 621
La Jolla CA 92093-1110

Colo State Univ, Boulder
1111 Engineering Dr
422 UCB
Boulder CO 80309

GTRI
505 10th St
Atlanta GA 30332

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson AFB, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency/
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2011-1310

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO case number; DISTAR 19483, Clearance Date: 11 June 2012. This report contains color.

14. ABSTRACT
Basic data flow patterns that we call performance idioms, such as stream, transpose, reduction, random access and stencil, are common in scientific numerical applications. We hypothesize that a small number of idioms can cover most programming constructs that dominate the execution time of scientific codes and can be used to approximate the application performance. To check these hypotheses, we built an automatic idioms recognition method tool and applied it to the UHPC Challenge problems.

15. SUBJECT TERMS
compiler, test, evaluation, source code, idioms, applications, architectures, stream, software, programs, programming, performance

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Kerry Hill
Unclassified	Unclassified	Unclassified	SAR	101	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 528-8897

Abstract

UCSD's goal under DARPA's Architecture-Aware Compiler Environment (AACE) program was to dramatically reduce application development costs and labor; ensure that executable code is optimal, correct, and timely; provide the full capabilities of computing system advances to our warfighters; and provide superior design and performance capabilities across a broad range of applications.

UCSD's work under AACE sought to develop productive, computationally efficient compilers and runtime systems for a broad spectrum of system configurations and applicable to a broad spectrum of DOD relevant applications. Compilers should be constructed based on a modular design, where the actual modules are selected and optimized based on the architectural characterization of a particular computing system. The overall process should result in an automatically self-assembling, optimized compiler that doesn't require user involvement or expertise. Therefore two Task 1 (T1) performers, Adaptive Environment for Supercomputing with Optimized and Parallelism (AESOP) and Platform Aware Compilation Environment (PACE) under the Defense Advanced Research Projects Agency (DARPA) Architecture Aware Compiler Environments Program were selected to develop these technologies. In addition, two Task 2 (T2) performers, Metrics for Architecture Aware Compiler Environments (MAACE) and Blackjack were chosen to measure progress of the T1s. This document describes work performed by the MAACE team.

Basic data flow patterns that we call performance idioms, such as stream, transpose, reduction, random access and stencil, are common in scientific numerical applications. We hypothesized that a small number of idioms can cover most programming constructs that dominate the execution time of scientific codes and can be used to approximate the application performance. To check these hypotheses, we built an automatic idioms recognition method tool and first applied it to comparing the performance of the Task 1 (T1) performers, AESOP and PACE; the toolsuite and methodology we developed was effective in evaluating the performers and teasing apart their differences. Suppose one has access to a computer and is considering porting code to take advantage of the special features? Or suppose one is evaluating a new compiler technology? Is there a reason to suppose the purchase cost or programmer effort will be worth it? We provide the ability to estimate the expected improvements in advance of paying money or time. The MAACE performer team exhibits an analytical framework and tool-set for providing such estimates: the tools first look for user-defined idioms that are patterns of computation and data access identified in advance as possibly being able to benefit from the new feature(s). A performance model is then applied to estimate how much faster these idioms would be if they were ported and run on the new machine or transformed by the new compiler, and a recommendation is made as to whether or not each idiom is worth the porting effort, and an estimate is provided of what the overall application speedup would be if this were done.

Table of Contents

List of Figures

Figure **Page**

List of Tables

Table **Page**

Foreword

This document describes work performed by the Metrics for Architecture Aware Compiler Environments (MAACE) team.

Preface

This work represents an advance in the state-of-the-art for automatic idiom recognition used for compiler and architecture and application comparison and evaluation.

Acknowledgements

This work was sponsored by DARPA. We wish to thank the San Diego Supercomputer Center for the use of their facilities.

1.0 Summary

The work of the MAACE performer team of the Architecture-Aware Compiler Environment (AACE) program has the goal to dramatically reduce application development costs and labor; ensure that executable code is optimal, correct, and timely; provide the full capabilities of computing system advances to our warfighters; and provide superior design and performance capabilities across a broad range of applications.

The MAACE team worked to develop productive, computationally efficient compilers and runtime systems for a broad spectrum of system configurations and applicable to a broad spectrum of DOD relevant applications. Compilers should be constructed based on a modular design, where the actual modules are selected and optimized based on the architectural characterization of a particular computing system. The overall process should result in an automatically self-assembling, optimized compiler that doesn't require user involvement or expertise. Therefore two Task 1 (T1) performers, Adaptive Environment for Supercomputing with Optimized and Parallelism (AESOP) and Platform Aware Compilation Environment (PACE) under the Defense Advanced Research Projects Agency (DARPA) Architecture Aware Compiler Environments Program were selected to develop these technologies.

The research program, known as AESOP, or Adaptive Environment for Supercompiling with Optimized Parallelism, represents a major multi-institution collaboration to develop a state-of-the-art compiler that can compile serial programs automatically into parallel programs to a wide variety of platforms. The collaboration includes the University of Maryland, BAE Systems Inc. and Princeton University. Reflecting the belief that serial programs will continue to represent the vast majority of programs in the world, the goal of the AESOP project is to compile serial programs automatically into parallel programs. Unlike existing efforts which have focused on regular, scientific programs alone, the AESOP project will use an aggressive suite of existing methods and new techniques that the researchers have developed to extract large-amounts of scalable parallelism even from seemingly serial irregular programs. This will enable software to exploit the full potential of the hardware in the modern multi-core era. Further, the compiler will accurately characterize and compile to a wide variety of computer systems without any manual effort.

PACE is an acronym for "platform-aware compilation environment." The PACE team includes researchers from Rice University, ET International, Ohio State University, Stanford University, and Texas Instruments.

In addition, two T2 performers, Metrics for Architecture Aware Compiler Environments (MAACE) and Blackjack were chosen to measure progress of the T1s.

The MAACE team is made up of researchers from the University of California, San Diego, (UCSD), University of Colorado, and Georgia Tech Research Institute. This document describes work performed by the MAACE) team.

The Blackjack team is made up of researchers from the University of Tennessee Knoxville (UTK), Oakridge National Laboratory (ORNL) and Rice University.

The MAACE T2 project was initially carried out through Phase I to evaluate the performance of T1 compiler developers PACE and AESOP as part of the DARPA Architecture Aware Environments (AACE) program and then, at the conclusion of Phase I, retargeted to develop a general idiom recognizer tool. The speed of the memory subsystem often constrains the performance of large-scale parallel applications. Experts tune such applications to use hierarchical memory subsystems efficiently. Hardware features such as accelerators, or multicore chips etc., can potentially improve memory performance beyond the capabilities of traditional hierarchical systems. However, the addition of such specialized hardware complicates code porting and tuning. During porting and tuning expert application engineers manually browse source code and identify memory access patterns that are candidates for optimization and tuning. HPC applications typically contain thousands to hundreds of thousands of lines of code, creating a labor intensive challenge for the expert. PIR, PMaC's Static Idiom Recognizer, was therefore developed to automate the pattern recognition process. PIR recognizes specified patterns and tags the source code where they appear using static analysis. This technology was developed and used to evaluate the DARPA AACE T1 performers and was retargeted then allowing us to add many new idioms to PIR, develop user and programmer manuals for it, and apply it to several applications of mission relevance including the UHPC Challenge Problems.

2.0 Introduction

With the advent of major changes in technology scaling that is the foundation of Moore's law, the continued advance of computing performance face major challenges [1, 2]. DARPA funded the Architecture Aware Compiler (AACE) Environments program under which this work (MAACE) was supported. Later, when AACE was canceled this funding was retargeted at developing idiom recognition capability that could benefit UHPC and other projects.

This document then provides the DARPA Architecture Aware Compiler Environments (AACE) final report for Task 2 (T2) Metrics for Architecture Aware Compiler Environments (MAACE) team and contains scores and explanation of scores for the Task 1 (T1) teams PACE and AESOP. We describe our scoring methodology, the Private test systems, and the overall scores. We find that modulo some bug fixes, both teams achieved 75%+ accuracy on the Private machines so we recommended a PASS for both AESOP and PACE in Phase 1.

We then detail the parameters that we used in the scoring formula and show the resulting scoring curves. This final report then describes our methodology for determining the truth value for each machine characteristic. This final report gives the detailed scores for PACE and for AESOP. We analyze the major discrepancies between our truth values and the measured values from PACE and AESOP.

The report goes on to describe the extensions to the Idiom Finding technology (PIR) that were the result of retargeting the funds in Phase II.

To focus and drive the UHPC research efforts with a long-term target (a six-year horizon for a technology demonstration system) and to ensure that the technology research focuses on the broad spectrum, the UHPC program includes five "Challenge Problems", complex applications chosen as exemplars of the diverse workloads that are important across a wide spectrum of system types / uses. For example, they include low-level, real-time sensor data processing and high-level decision support. Though computational modeling (scientific computing) is well represented, areas of emerging importance such as graph algorithms are also included. The challenge problems include: Streaming Sensor, Dynamic Graph, Chess, Molecular dynamics, and Shock Hydrodynamics. This spectrum is broader than most benchmark suites, and is chosen to reflect a variety of leading edge and future information and computational needs for the US Department of Defense, National Intelligence, and Homeland Security. While a number of these application codes are new, several are derived from combinations of existing codes. Further, an evaluation team is included in the program with the charter to drive the development of the challenge problems and also to measure the progress of UHPC technology.

We developed a set of tools for idiom recognition. Then we used them to analyze the five challenge problems empirically, exploring their scaling properties, computation and datatype needs, memory behavior, and temporal behavior. These empirical studies form an initial baseline for requirements and opportunities for Exascale class computer architecture and software. They also allow an initial validation of the PIR toolset and the application assumptions underlying the Exascale studies [3, 4].

The detailed characteristics of the challenge problems also bear on opportunities for exploiting customized architectures for greater performance and energy efficiency. Researchers have proposed a new paradigm, "10x10" [5], which involve the systematic analysis, assessment, and design of heterogeneous hardware approaches for general-purpose computing. By setting the architect and software developer in a multi-polar framework, 10x10 enables the full benefits of hardware customization to be tapped. The key idea in 10x10 is to classify application phases into (nominally) 10 distinct clusters, each defined by common characteristics (data types, operations, algorithms, etc.). Though each cluster represents a small part of the overall general-purpose workload (perhaps 10%), clustering exposes opportunities for architecture and implementation customization. Modular composition of the customized architectures captures the energy efficiency and performance benefits.

In that vein, we also examine the results as a first step towards an application classification for a 10x10 architecture [1,5] by highlighting the opportunities for customization for greater efficiency in execution. In some cases they show behavior similar to that assumed in the Exascale studies; in other cases, there is divergence.

Contributions of this MAACE project including AACE Phase I and the retargeted Phase II include:

- An exhibition of the methods used to evaluate architecture aware compiler developers
- A concise description of the UHPC challenge problems, and characterization of their basic scaling properties
- Characterizing the compute idioms [18] exercised by the challenge problems, showing that both regular and irregular access structures are important for UHPC compute engines,
- Characterizing the data, operation, instruction, data types and sizes, used in each challenge problem, showing that micro-architecture designers have many opportunities to exploit heterogeneous simplification for greater efficiency,
- Characterizing the memory behavior of the UHPC applications to show that four of the five challenge problems exhibit sufficient data locality, enabling DRAM energy to be kept within "exascale strawman" energy projections, but one does not.
- Identification of challenge problem characteristics that are opportunities to exploit heterogeneity for performance and energy efficiency that form a basis for initial 10x10 application clustering.

3.0 Methods, Assumptions, and Procedures

3.1 Scoring Methodology

For evaluation purposes we use the probability density function of the Weibull distribution, where the scale parameter is derived from the provided shape parameter in order to center the median at the "truth value". Weibull distributions were selected in order to provide a simple, but effective means to control the tolerance to errors in characterization. Control over tolerance to errors allows us to tailor the scoring to match the expected importance of each characteristic to the overall compiler performance. We compute the score based on the calculation of (Wpdf(true) – Wpdf(measured))/Wpdf(true) where true and measured are values of some computer characteristics. The scoring formula uses the root mean square and weights attached to each truth value as follows

3.1.1 Equation 1: Score

$$\frac{\left[\sum_{i=1}^{Chars.} Weight_i\right] - \sqrt{\sum_{i=1}^{Chars.} \left(Weight_i \frac{Wpdf_i(correct_i) - Wpdf_i(measured_i)}{Wpdf_i(correct_i)}\right)^2}}{\sum_{i=1}^{Chars.} Weight_i}$$

The above formula gives a deliberate incentive to report more values. The score tends to go up with quantity (more characteristics reported) as well as quality (more accurate characteristics). It was designed to encourage Task 1 teams to report many characteristics. In retrospect, the above formula is too lenient with respect to quality.

Therefore, to isolate quality, we also provide a formula based simply on absolute error as follows

3.1.2 Equation 2: Quality

$$\frac{\sum_{i=1}^{Chars.} Weight_i - \sum_{i=1}^{Chars.} \sqrt{Weight_i \left(\frac{Wpdf_i(correct_i) - Wpdf_i(measured_i)}{Wpdf_i(correct_i)}\right)^2}}{\sum_{i=1}^{Chars.} Weight_i}$$

The difference is whether the summation is inside (Score) or outside the radical (Quality). While Score incentivizes providing a greater number of characterizations, Quality provides a finer discrimination of characterization accuracy, with errors in the value of individual characteristics yielding a more visible penalty. Since both T1 teams are providing a large array of characteristics now, and the accuracy required for Phase 2 would have been increasingly important, we intended to use Quality as the official scoring formula for Phase 2 had there been a Phase 2. For the purposes of this Phase 1 report however Score is the official scoring formula and Quality is metric related to qualitative analysis of Score that will be used in the discussions that follow below.

To generate each $Wpdf_i$, we select a shape parameter (larger yields a steeper penalty for being wrong), then we compute scale = (Truth Value) / (ln(2)**(1/shape)) and use that in the Weibull function generator.

3.1.3 Equation 3

$$Wpdf_i(x) = \frac{shape}{scale}\left(\frac{x}{scale}\right)^{shape-1} e^{-\left(\frac{x}{scale}\right)^{shape}}$$

yielding a Weibull function centered around the "truth value" and of the chosen appropriate shape

Additionally, we introduce an optional symmetry parameter such that symmetry = -1 flips $Wpdf_i$ about its center (see examples below) and that allows us to have functions that penalizes less for being wrong on the low side as opposed to less for being wrong on the high side (the default behavior).

3.1.4 Equation 4

$$\frac{\left[\sum_{i=1}^{Chars.} Weight_i\right] - \sqrt{\sum_{i=1}^{Chars.}\left(Weight_i \frac{Wpdf_i(correct_i) - Wpdf_i(2(correct_i) - measured_i)}{Wpdf_i(correct_i)}\right)^2}}{\sum_{i=1}^{Chars.} Weight_i}$$

A similar symmetry flip can be used in the Quality formula.

We determined the correct "truth values" via independent analysis and measurement of the machines described in the next section. We determined the weights (the importance of each characteristic) based on advice from the T1 teams regarding which characteristics they deemed most important, and also using our own experience to reflect what we deem to be most important for an optimizing compiler to know about the target platform.

3.2 Private Systems

We evaluated the characterization tool provided by each of the two T1 teams using PIR on 3 so-called Private systems, which were unknown to the T1 teams. We characterized the Private systems independently to determine the correct "truth values" for the set of characteristics that the T1 teams asked to be scored on. The three Private systems are AACE-PS3 a Play Station cluster, SDR a Xeon Cluster, and Triton, another Xeon clusters. We provided the T1 and T2 teams' access to 3 Public systems, along with our "truth values" on those systems to enable them to calibrate their methods and to allow iteration leading to better consensus regarding the semantics of the characteristics. The 3 **Public Systems** were Dash, a Nehalem cluster at SDSC, Batcave, an SGI Altix at SDSC, and an ARM system at Colorado.

The 3 **Private** systems (those used in the scoring) are described in detail next.

3.2.1 AACE-PS3

AACE-PS3 is a PlayStation 3 machine with Yellow Dog Linux as its operating system and a CELL microprocessor running at a clock-rate of 3.3 GHz. The processor consists of one Power-Processing Element (PPE) and eight fully functional co-processors called Synergistic Processing Element (SPE). Both the SPE and the PPE access system memory through a shared memory interface controller. Figure 1 shows the components of a CELL processor.

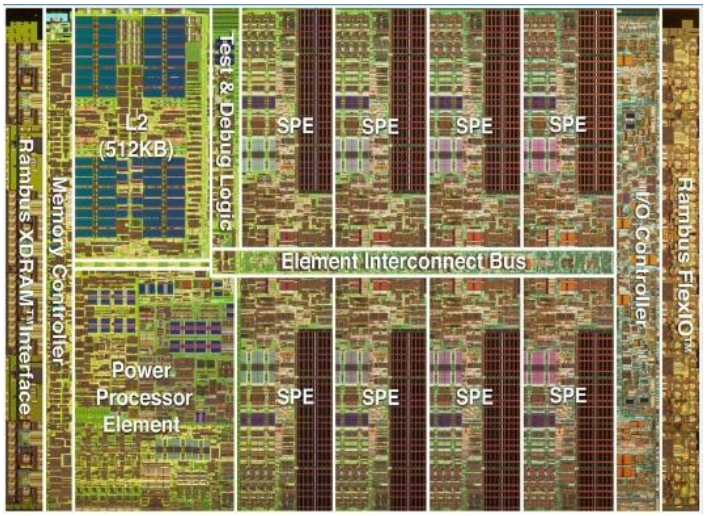

Figure 1: CELL Processor Die-Micrograph

Our test system only makes use of the PPE processor; the SPE processors are ignored. The PPE is a 32-bit two-way SMT multithreaded PowerPC based core. There are two sets of 64-bit register files to store floating-point and fixed-point values. Some 64 bit operations are performed on the SPE, whereas all the 32 bit computations and I/O are performed on the PPE. Each PPE has two-levels of cache at 32 KB and 512 KB, respectively. In this project, the CELL processor is used to study how well the characterization tools perform on a POWER6 architecture.

3.2.2 SDR

SDR is a 16-node cluster located at the University of Colorado. Each node consists of two quad-core Intel Xeon E5450 processors running at 3 GHz. The nodes are connected to each other using a high-bandwidth (DDR 4x 20Gbps) Infiniband network. Intel Xeon E5450 is a CISC out of order processor with two-levels of cache. L1 is 256 KB and L2 is 12 MB. Each node is also connected to 16 GB RAM.

3.2.3 Triton

The Triton compute cluster is a Appro Hypergreen cluster based on Intel Xeon processor 5500 series. The cluster features Appro gB222X Blade server nodes with dual quad-core Intel Xeon E5530 processors with Intel Nehalem micro-architecture, running at 2.4 GHz clock-speed. Each of the 256 nodes has 24 GB of memory and an 8 MB cache. In addition, there are four end-nodes to handle all administrative tasks. Figure 2 shows the die-micrograph of a quad-core Xeon E5530 processor.

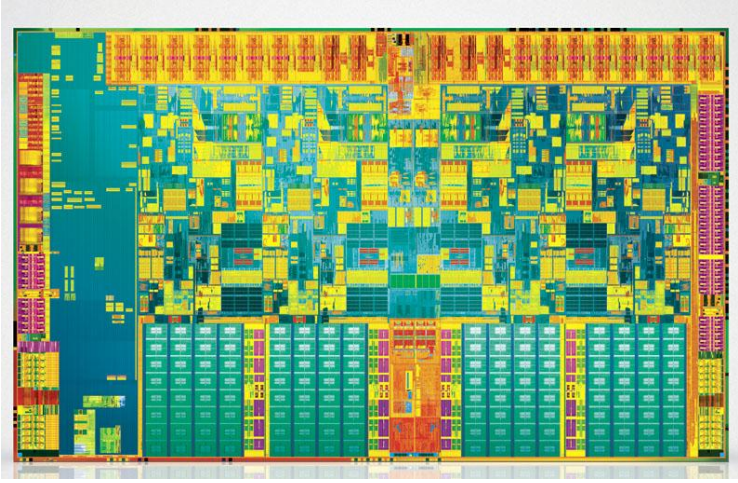

Figure 2: Intel Xeon 5500 Series Micrograph

Each node has a 10-GB Myrinet connection, giving the system a total bandwidth capacity of 256 GB/s. The cluster has a peak theoretical throughput of approximately 20 teraflops and contains 6 TB of memory. In addition, the cluster utilizes 90% efficient power supplies and 20% power reduction when compared to standard rack servers.

3.3 Tools Used in Evaluation

PIR (PMaC Idiom Recognizer) is a tool for searching source code for idioms.

An idiom is a local pattern of computation that a user may expect to occur frequently in certain applications. For example, a stream idiom is a pattern where memory is read from an array, some computation may be done on this data, and then the data is written to another array. A stream reads sequentially from the source array and writes sequentially to the destination array. A stream may arise from the presence of the statement A[i] = B[i] within a loop over i.

Idioms are useful for describing patterns of computation that have the potential to be optimized, for example, by offloading the piece of code to a coprocessor or GPU.

The PIR tool allows us to automate searching for idioms in a powerful way by using data-flow analysis to augment the identification process. It would be very difficult to use a simpler searching tool, such as regular expressions, because a regular expression does not naturally discern the meaning of the text it identifies. For example, in the code

```
1       values[c] = constants[c];
2
3       for( i = 0; i < 10; ++i ) {
4       item = source array[i];
5       dest_array[i] = item;
6       }
```

A simple regular expression that searches for stream idioms of the form A[i] = B[i] would incorrectly identify line 1 and it would miss the stream at lines 4-5 because the assignment is broken into multiple statements.

PIR, however, is able to determine that line 1 is not in a loop and that c is a constant. This indicates that the meaning of this statement is simply a variable assignment, rather than a stream. In lines 4-5, PIR uses data-flow analysis to determine that item in line 5 holds a value from the source array making this a stream.

3.4 Codes Used in Evaluation

To focus and drive the UHPC research efforts, the program includes five "Challenge Problems", complex applications chosen as exemplars of the diverse workloads that are important across a wide spectrum of system types / uses. These codes were chosen for diversity and their relevance to critical government needs. The challenge problems include: Streaming Sensor, Dynamic Graph, Chess, Molecular dynamics, and Shock Hydrodynamics. Of the 5 Dynamic Graph and Streaming Sensor are deemed to be the best representatives of DoD mission-critical applications i.e. real-time surveillance (Streaming Sensor) and advanced analytics (Dynamic Graph). We wanted to see if the requirements of ordinary commercial workloads exemplify (or not) the requirements of DoD. We applied PIR to the study of these problems. In the following text, we briefly describe each of the five UHPC challenge problems:

3.5 Streaming Sensor

The Streaming Sensor Challenge Problem (SSCP) models a wide-area, high resolution persistent surveillance mission. The problem is based around radar image formation and analysis for knowledge extraction, representing the processing operations required to transform a stream of inputs from a sensor suite associated with a radar into a set of possible detections of moving objects for further tracking and analysis. The persistent surveillance mission is characterized by having one or more sensors continually monitoring a region of interest. In the SSCP, a synthetic aperture radar (SAR) is the sensor of primary interest. The SAR produces fine-resolution imagery which is combined with images taken at different times in order to create a composite image indicating the locations of any changes that may have occurred within the scene. The SSCP requires that SAR image formation be accomplished via backprojection [6].

In the mission modeled by the SSCP, an airborne radar system flies a repeated path around a target area to be observed. The radar illuminates the target area with regular, repeated radar pulses. The primary sensor inputs for the challenge problem are the complex-valued return samples, the time of transmission of each pulse, and the position of the transceiver at the transmission time of each pulse.

Successive full-size images must be constructed at a specified cadence using overlapping subsets of pulses. Those images taken from the same nominal position from consecutive orbits are then registered via a two stage process comprising a global affine transformation for coarse registration followed by a thin plate spline warping for fine local registration. Once registered, coherent change detection (CCD) between successive images is applied, followed by a constant false alarm rate (CFAR) algorithm to identify pixel locations of significant change.

3.6 Dynamic Graph

The Dynamic Graph problem models a growing class of graph-based applications emerging in chemistry, biology, medicine, social sciences, and security applications. Challenges include extremely large graphs (the Facebook friend network has over 500 million users with an average of

130 connections [7], and Twitter sends 140 million messages per day [8]). Graphs originating from across this spectrum often exhibit common characteristics such as a low diameter and a power-law distribution in the number of neighbors [9], making balanced graph partitioning for parallel computation difficult.

Dynamic graph includes four kernels: The first two generate the graphs. First, a data generator produces an initial graph and stream of edges using R-MAT, the recursive matrix algorithm [10]. The second kernel converts the graph into a specialized "stinger" data structure, which provides a good trade-off between speed, parallelism, and memory size.

The last two kernels analyze the streaming graph with respect to two global properties. One kernel calculates the number of connected components, producing for each component, the number of vertices and edges in it. This is a fundamental property important for many higher level analyses. The last kernel identifies subgraphs that locally optimize some connectivity criterion. This kernel uses a dynamic variant of an agglomerative algorithm and its results can be important for visualization, data access, a disjoint partitioning for gene classification, organization structure, metabolic structure and others. Dynamic streaming graph analysis taxes current hardware and software architectures due to the size and structure of typical inputs.

3.7 Chess

The Chess problem is an exemplar for complex decision-support and search applications. The computational complexity of Chess is well understood, and high capability depends on artful blending of domain knowledge and heuristic based approximate results [11,12]. The chess game tree exhibits rapidly changing directed-graph structures that grow exponentially with depth. The game tree is pruned dynamically based on an objective function consistent with a heuristic and domain knowledge. Ideally, heuristic approaches eliminate paths that do not contribute towards the final outcome. In practice, pruning must be carefully tuned based on domain knowledge.

The Chess challenge problem is to estimate the best next move that can be computed within a two hour deadline. The score of the challenge problem consists of the ply achieved, plus 120 minus the number of minutes required to achieve that result. Board positions are to be evaluated using the MTD-f algorithm [13] and a quiescent search that explores captures (except en passant). To make the next move selected more deterministic, the Zobrist hash [14] is combined with heuristic score. This requires that a "random" number needed for the Zobrist be saved and re-used every time the benchmark is run.

3.8 Molecular Dynamics

The UHPC Molecular Dynamics Challenge Problem simulates the atom-by-atom interactions between molecules in order to derive macroscopic properties of materials and to understand microscopic atomic-level phenomena that cannot be observed directly. There are numerous well-known software packages for Molecular dynamics [15], including GROMACS, NAMD, AMBER, and LAMMPS. The challenge problem is based on a simplified sequential version of LAMMPS [16].

3.9 Shock Hydrodynamics

Computer simulations of a wide variety of science and engineering problems require modeling hydrodynamics, which describes the motion of materials relative to each other when subject to forces. Many important DoD simulation problems involve complex multi-material systems that undergo large deformations. Examples include armor defense, penetration mechanics, blast effects, structural integrity, and conventional munitions such as shaped charges and explosively formed projectiles.

Hydrocodes typically partition the spatial problem domain into a collection of volumetric elements defined by a mesh. A node on the mesh is a point where mesh lines intersect. Finite difference equations that approximate differential operators in the equations couple variables on the mesh (e.g., at nodes and elements) via stencil operations. Other computations, involving material properties and equation of state, are interleaved with the stencil operations. The operations must be performed in a specific order for numerical accuracy and computational robustness.

The shock hydrodynamics challenge problem models high deformation events via Lagrangian shock hydrodynamics.

In Lagrangian hydrocodes, the initial mesh configuration partitions the problem domain into material elements and element boundaries are constructed to align with material interfaces. As a simulation evolves, the mesh follows the motion of these elements through space and time. Lagrangian methods handle moving boundaries and multiple materials naturally and can provide a highly accurate solution for many problems without requiring an excessively fine mesh. However, when the flow involves sufficiently complex structure (e.g., strong shearing or vorticity), Lagrangian methods can perform poorly as mesh elements distort and possibly tangle.

The shock hydrodynamics challenge problem is a greatly simplified derivation from ALE3D [17]. The challenge problem solves the Sedov blast wave problem for one material in three dimensions. The problem has an analytic solution, and can be scaled to arbitrarily large problem sizes.

ALE (Arbitrary Lagrangian Eulerian) codes (such as ALE3D [17], an LLNL code used by DoD) have been developed to seek a compromise between the Eulerian and Lagrangian formulations. ALE methods can accurately solve problems involving moving boundaries, multiple materials, and strong shearing and vortical flow regions. The general strategy is to evolve the problem using the Lagrangian algorithm until the mesh reaches a level of distortion such that continuing in this fashion is problematic. At this point, the mesh is relaxed to a more numerically-desirable configuration. Then, the simulation variables are mapped to the new mesh and the simulation continues.

4.0 Results and Discussion

4.1 Scores

4.1.1 AESOP

The following table summarizes Score and Quality across the different Public and Private systems for AESOP.

Table 1: Score and Quality - AESOP

	System	Score	Quality
Public	DASH	99%	97%
	Batcave	96%	89%
	ARM	99%	98%
Private	PS3-PPE	97%	90%
	Triton	99%	97%
	SDR	95%	86%

All of the Score values are at the upper end of the acceptable range and Quality is also high (75%+), resulting in a strong PASS for AESOP for Phase 1. It is interesting to note that Quality is not significantly lower than Score, and the Score on the Private systems is comparable to that on the Public Systems. The AESOP characterization tool delivers both quality and quantity.

4.1.2 PACE

The following table summarizes the Score and Quality metrics for the different Public and Private systems for PACE.

Table 2: Score and Quality – PACE

	System	Score	Quality
Public	DASH	93%	73%
	Batcave	97%	91%
	ARM	97%	95%
Private	PS3-PPE	94%	76%
	Triton	94%	75%
	SDR	93%	72%

All of the Score values are at the upper end of the acceptable range but the Quality is in some cases marginal (below 75%). It is interesting to note a significant gap between their Quality and Score *even on Public systems* – that is to say even on systems where there was ample time to discover such deficiencies in advance. PACE has more quantity than quality—they characterize a lot of values but not all of them are accurate. During testing we discovered a trivial programming bug in a few of PACE's benchmarks (described in Section 0 on page 39) that resulted in unintended and incorrect values being reported by the PACE characterization tools on some architectures (including Public systems). After we implemented a fix for this bug, PACE's machine scores improved to those in the following table.

Table 3: Score and Quality, Post Fix – PACE

	System	Score	Quality
Public	DASH	95%	85%
	Batcave	97%	92%
	ARM	97%	94%

Private	PS3-PPE	94%	77%
	Triton	95%	87%
	SDR	94%	84%

These scores demonstrate that PACE's overall methodology is sound and that the coding error significantly lowered Quality for their characterization tool. The result is a Pass for PACE in Phase 1.

4.2 Parameters Used in Results

The following are the MAACE-assigned weights and Weibull parameters for the characteristics measured by the T1 teams.

Table 4: MAACE Assigned Weights and Weibull Parameters – PACE

Characteristic	Units	Weight	Shape (k)	Symm
L1 cache size	Bytes	3	5	-1
L2 cache size	Bytes	3	5	-1
L3 cache size	Bytes	1	4	-1
L1 TLB size	Bytes	2	3	-1
L2 TLB size	Bytes	2	3	-1
L1 line size	Bytes	3	5	-1
L2 line size	Bytes	3	5	-1
L3 line size	Bytes	1	4	-1
TLB page size	Bytes	2	5	-1
L1 associativity	Integer	3	5	-1
L1 latency	Ratio to integer add	3	5	1
L2 latency	Ratio to integer add	3	5	1
L3 latency	Ratio to integer add	1	4	1
Maximum simultaneous operations[i]	Integer	3	5	1
Operation costs[ii]	Ratio to integer add	3	3	1
Maximum int32 live values	Integer	2	5	1
Maximum int64 live values	Integer	2	5	1

Maximum flt32 live values	Integer	2	5	1
Maximum flt64 live values	Integer	2	5	1
Effective integer contexts	Integer	2	5	1
Effective float contexts	Integer	2	5	1
Effective memory contexts	Integer	2	5	1

Table 5: MAACE Assigned Weights and Weibull Parameters – AESOP

Characteristic	**Units**	**Weight**	**Shape (k)**	**Symm**
L1 cache size	Bytes	3	4	-1
L2 cache size	Bytes	2	4	-1
L1 line size	Bytes	3	4	-1
L2 line size	Bytes	2	4	-1
L1 associativity	Integer	3	4	-1
L2 associativity	Integer	2	4	-1
Operation throughputs[iii]	Ops per integer add	3	5	1
Operation costs[iv]	Ratio to integer add	3	3	1
Effective integer contexts	Integer	3	5	1
Effective float contexts	Integer	3	5	1
Effective memory contexts	Integer	3	5	1
NUMA node count	Integer	1	3	1
NUMA node size	Integer	1	3	1

i This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}
ii This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}
iii This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}
iv This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}

4.3 Demonstration of Scoring Curves and Discussion

The following plots show scoring curves based on our scoring definition, for various shape parameters (which are shown in the plots as *k*). For these plots, the truth value is chosen to be 1, since any truth value/measured pair can be normalized to 1. The x-values for the displayed points from left to right are: 0.25, 0.5, 0.75, 0.9, 1.1, 1.25, 1.5, 1.75. These correspond to the cases where the measured value is 25%, 50%, 75%, 90%, 110%, 125%, 150%, 175% of the truth value. For skew clarity, blue dots are the over-estimated values and red the under-estimated values.

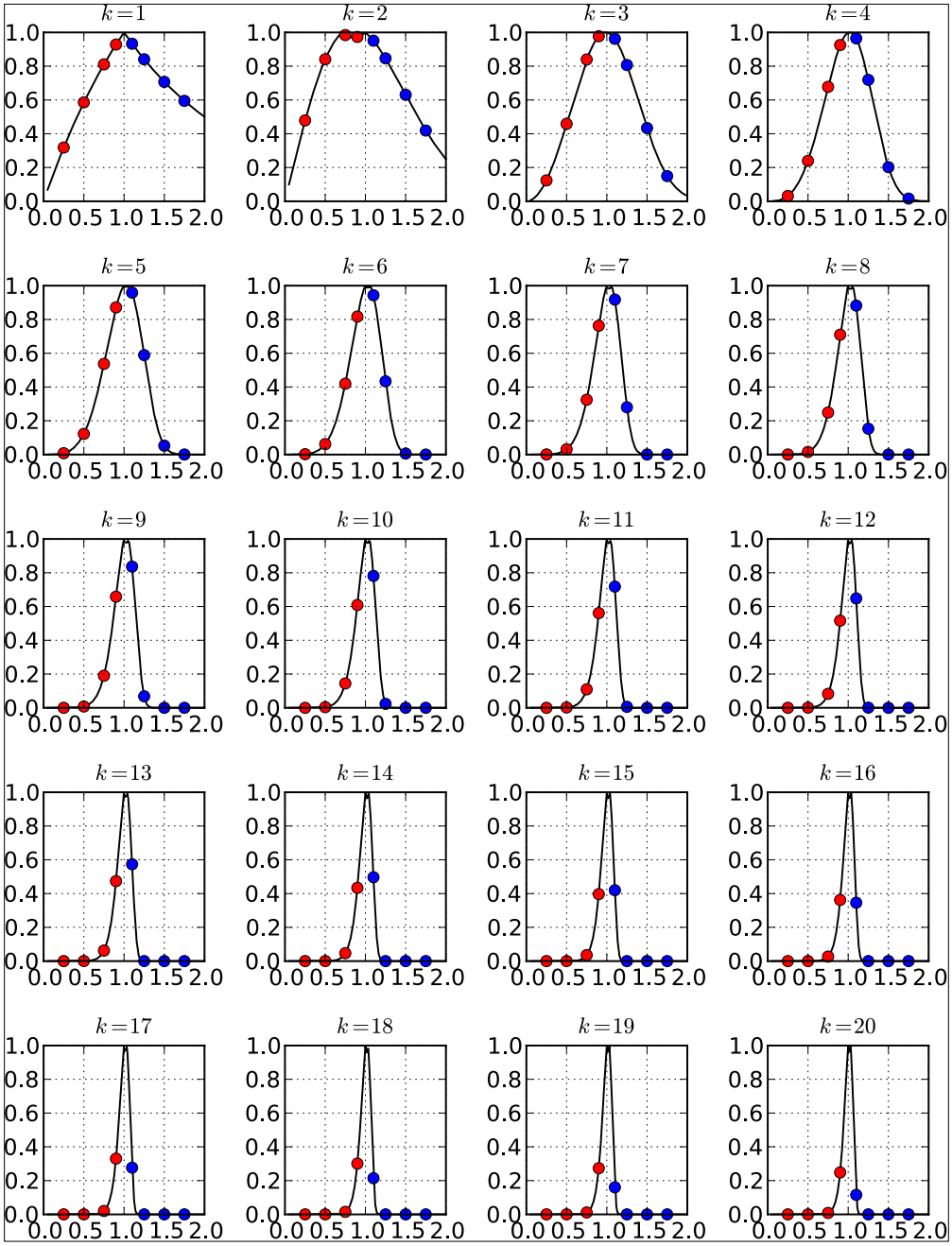

Figure 3: Scoring Curves

The following plots show the same information with higher resolution for four of the shapes. The displayed points show the measured/truth values of 10%, 20%, …, 90%, 110%, 120%, …, 190%. These all have symmetry 1 (positive) and show a bias (lower penalty) for being wrong on the high side (the blue dots are higher than the corresponding blue dots). Symmetry = -1 would obtain the opposite effect.

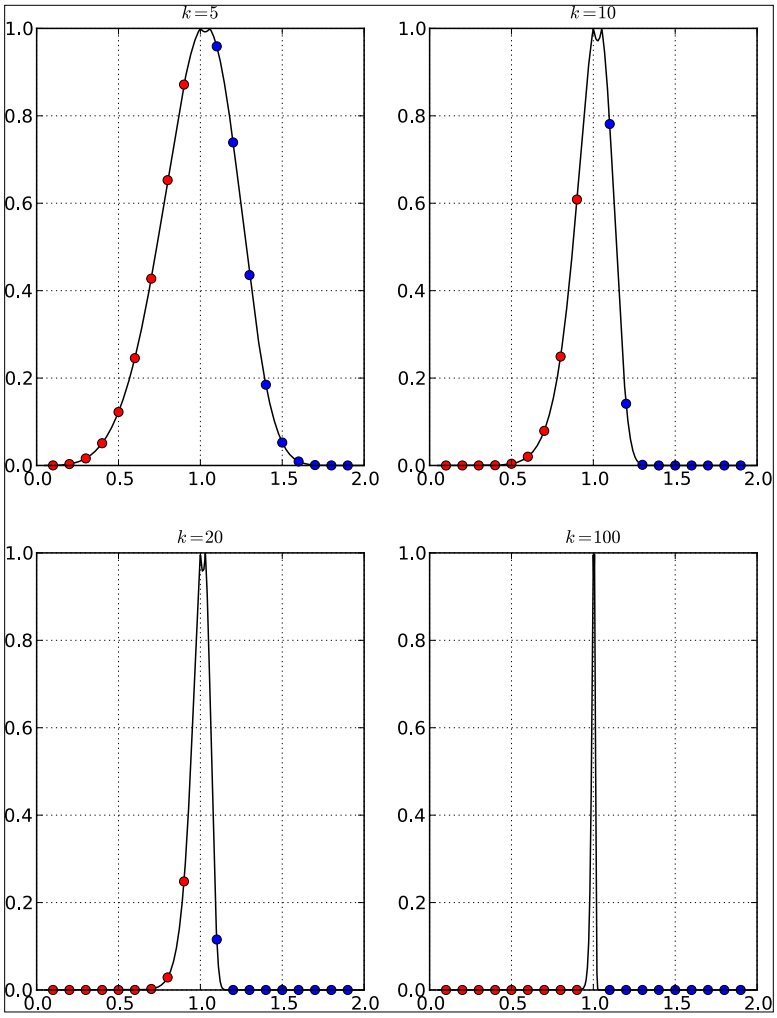

Figure 4: Scoring Curves – Higher Resolution

4.4 Discussion of Truth Values

This section describes how we determined truth values for the public and private systems. In many cases, values were determined using custom benchmarks developed specifically for the AACE project. Moreover, many of these benchmarks were continuously refined during phase 1 based on feedback from the T1 groups concerning the precise semantics of individual characteristics.

4.4.1 Cache Size

The L1, L2 and L3 cache sizes and cache-line sizes were determined using the Multi-MAPS (Machine Access Pattern Signature) benchmark probe. Multi-MAPS is a well-known benchmark derived from the STREAM benchmark that is designed to measure platform specific bandwidths. It is used in DoD TI-XX procurements and other agency performance modeling work. The measurements include bandwidths of different levels of memory, depending on argument and operation type as shown in **Figure 5** below. Multi-MAPS can be obtained from PMaC labs at SDSC.

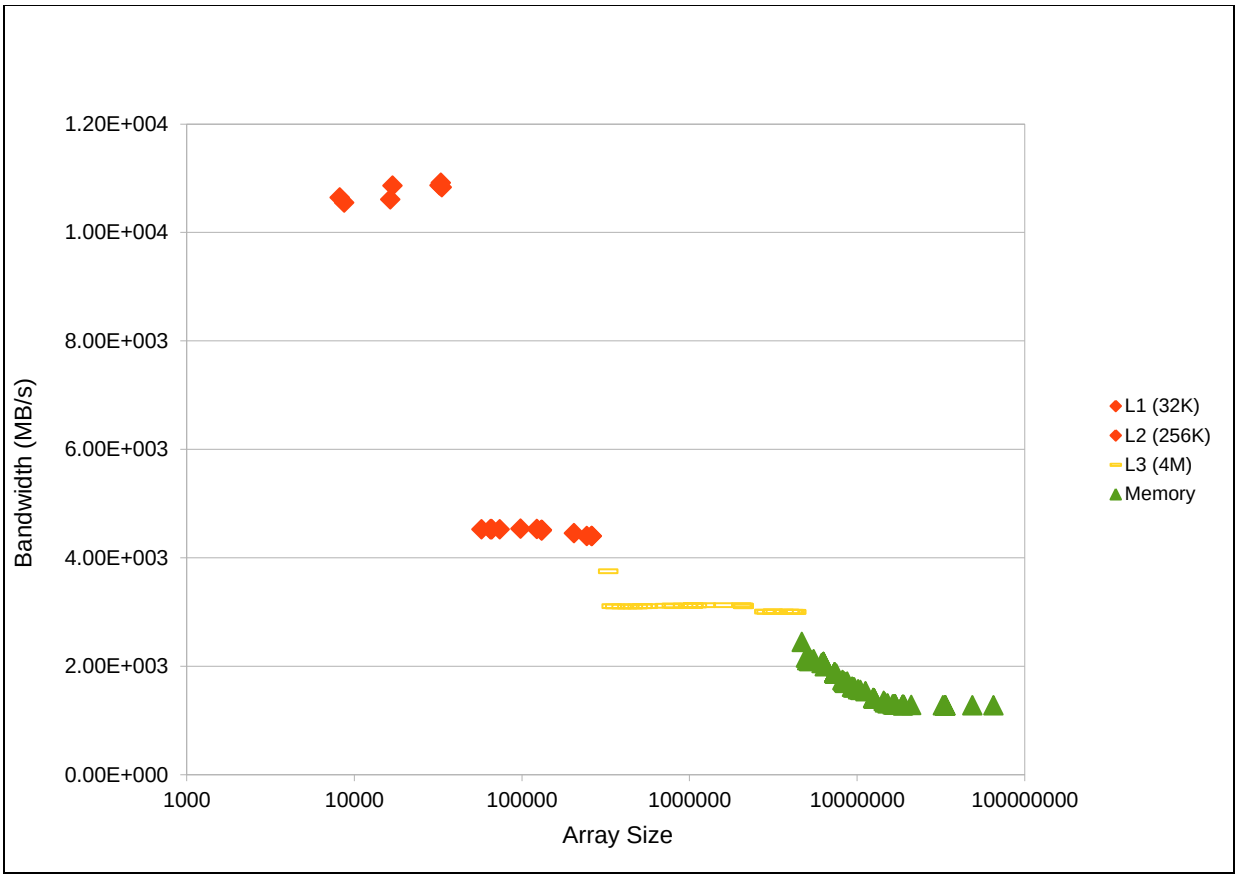

Figure 5: Dash Stride -8Memory Bandwidth
Note: Clear L1, L2, L3 (yellow) and Memory (green) regions

In general there was some debate about exactly where to draw the line between cache regions. The manufacturer specification may state a certain cache size but applications may begin to experience increased cache miss rates as its working set approaches that size. By the same token, applications may still benefit from a cache even if it has slightly exceeded the cache capacity. We checked the manufacturer specifications and then verified those values with the Multi-Maps tool. Because the T1 teams do not have the luxury of checking manufacturer specification on the Private machines, and because a compiler optimization may still improve performance if the cache size is characterized "about right" we used relatively wide Weibull's for cache size.

4.4.2 Cache Latency

We determined cache latency values using a combination of a custom benchmark and the **lat_mem_rd** benchmark from the open-source lmbench suite. The **lat_mem_rd** benchmark works well at computing read latencies for smaller caches in the cache hierarchy, but fails when probing caches that hold more data than can be referenced by the TLB. This limitation required us to develop a custom benchmark to compute latency values for large L2 and L3 caches.

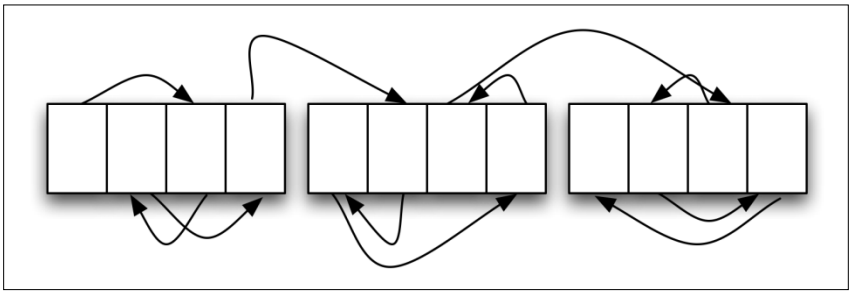

Figure 6 - Sample Access Pattern in Memory Analysis Tool

Our cache latency benchmark divides the evaluated cache into sub-blocks that are guaranteed to fit into the TLB. This is shown schematically in **Figure 6**, above. In that diagram, we assume the TLB can map four pages at the same time. Each page contains multiple cache lines (not shown). These sub-blocks are then tested individually, and a composite latency value is computed. To test a sub-block, the benchmark performs a random traversal that touches each cache line exactly once. Using a random traversal prevents the hardware prefetcher from fetching future values into lower cache levels, while hitting a cache line only once ensures that we never read a value that was placed into a lower level cache based on a previous access. Once all the cache lines in a given set of pages have been traversed, the measurement system moves to the next set of pages that can be simultaneously mapped in the TLB. This minimizes the number of TLB misses that occur; the delay from the remaining TLB misses is amortized over all the cache lines being measured in that collection of pages. Our custom benchmark matches the **lat_mem_rd** results for smaller cache sizes, while providing more accurate results for larger cache sizes.

4.4.3 Cache Associativity and Line Size

We determined both cache associativity and line size values based on the system specifications. On x86 machines such as DASH, SDR, and Triton, we used the values reported by the CPUID instruction. For all other machines, we used the values from the published machine specifications.

4.4.4 TLB Size and Page Size

For TLB page size, we determined the truth value using the number reported by the operating system. For all of our machines, this was accomplished using the getpagesize() library call. This approach is necessary because many architectures support multiple page sizes, and the page size in use is selected by the operating system.

For TLB size, we used the published values for DASH, Batcave, Triton, and SDR. For PS3-PPE and ARM, we were unable to locate a reliable source of published information concerning the TLB size. Therefore, we wrote a characterization benchmark to measure these values on ARM and PS3-PPE.

Our benchmark works by ensuring that a set of data is resident in the L1 cache in order to remove memory system effects and therefore isolate TLB behavior. The benchmark accesses a set of cache lines from different pages. The number of cache lines touched is designed to fit within the L1 cache, however the number of pages references should predictably saturate the different levels of TLB. By continually increasing the number of pages referenced, we are able to determine the inflection points that indicate the TLB sizes.

4.4.5 Operation Throughputs and Simultaneous Operations

We wrote a single benchmark for measuring both operation throughputs and simultaneous operations. The benchmark does not directly report these characteristics, but instead reports some data for each type and operation. We manually analyze the data produced by the benchmark to determine values for operation throughputs and simultaneous operations.

For each type and operation, the benchmark measures the time it takes to execute 1 through 15 streams of operations and reports work done over time in units of operations per integer addition for each case. To elaborate, it first measures the time to execute a single stream of operations. By "stream" we mean a sequence of dependent operations: each operation within the stream directly depends on one or two operations immediately before it. The benchmark then computes work done over time for this case. Next, it measures the time to execute two interleaved streams of operations. Each operation within either stream is dependent on operations in the same stream but independent of all operations in the other stream. If the hardware supports execution of two operations (of the type in consideration) in parallel, then the two streams should execute in parallel. Following the measurement, the benchmark computes work done over time for this case. Note that if the hardware executed the two streams in parallel, then the work done over time for this case should be higher than the single stream case. The benchmark performs the same steps for 3 through 15 streams.

Special care was taken to ensure that the operands to integer division operations do not degenerate to trivial values. This was done to counter-act implementations of some operations

(multiplication, division, transcendental functions, *etc.*) that use an "early exit" algorithm. For integer division operation (of either width), the benchmark starts with two distinct numbers for every stream. The larger of these numbers is a multiple of the smaller number. The first operation in every stream divides the larger number by the smaller number, producing a quotient that is also a factor of the larger number. The second operation in every stream divides the larger number by the result of the first operation, which gives back the smaller of the original numbers. All further operations divide the larger number by the result of the previous operation, continuing what we call "ping-ponging" of values. We have implemented a similar solution for floating-point division operations.

We performed some manual analysis on the data reported by the benchmark to determine values for throughput and simultaneous operations for every type and operation. For throughput, we simply went through all the work done over time for values pertaining to the type and operation in consideration and selected the maximum value. For simultaneous operations, we picked the smallest number of streams that gives the throughput value, ignoring any noise. For example, consider the plot in **Figure 7**, which shows the data collected by our benchmark for PS3-PPE flt32 add. The maximum work done is 2.0 operations per integer addition, which we selected as the throughput for PS3-PPE flt32 add. The smallest number of streams that gives the throughput is 10, which was selected as the maximum number of simultaneous PS3-PPE flt32 add operations.

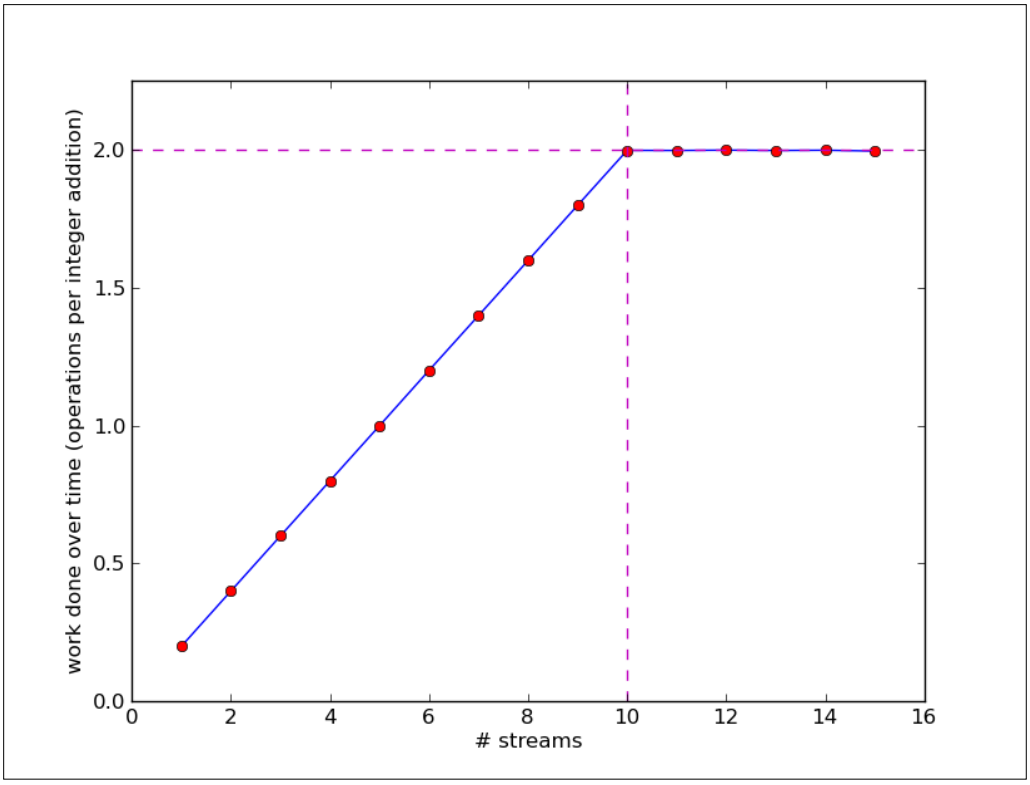

Figure 7 - Data collected by our benchmark for measuring operation throughputs and simultaneous operations for PS3-PPE flt add

4.4.6 Operation Latency/Cost

We developed two benchmarks to compute operation latency. This section discusses the general benchmark used on all of our machines except our public ARM machine. The next section discusses the ARM-specific benchmark.

Figure 8 shows the overall algorithm for the operation latency benchmark. The "for-loop" is run for a certain number of iterations as determined by the person running the test. For this test, the body of the for-loop was executed 100,000 times. We passed this number as a command line argument so that the compiler does not apply any cyclic-code optimizations.

```
gettimeofday(&start_time, NULL);

for-each (ii = 0 to argv[1])
{
  return_value = OPERATION_FUNC(return_value);
}

gettimeofday(&end_time, NULL);


totalTime = end_time - start_time
```

Figure 8 - Operation Latency Top-level Algorithm

In **Figure 8,** the "OPERATION_FUNC" is the function that performs either addition, subtraction, multiplication or division for 8-bit, 16-bit, 32-bit or 64-bit fixed-point or floating point operations. Inside this function, there are 12,500 operations. We varied the iteration size and the number of operations inside the "OPERATION_FUNC" and found that the operation cost in all machines seems to stay constant at 100,000 and 12,500 or greater, respectively. To prevent the compiler from parallelizing the different stages of the loop, we created inter-loop dependency by using the return value as the inputs to the operands.

For division, we found that for large numbers of divisions performed in this fashion, there is a high probability that the quotient will converge to zero. When the dividend is zero, the divide operation is not performed and a zero is returned. If the divisor is zero, then a "divide-by-zero" exception is returned. To avoid these two scenarios, we keep the dividend constant, and use the quotient from the last division operation, as the divisor of current operation. For example, for the first computation let us say the dividend is 55 and the divisor is 11. For the second computation, we use 5 (55/11 = 5) as the divisor and 55 as the dividend. This will create a dependency between the two operations, and avoids the above two problems.

4.5 Operation cost for ARM

The characterization of the ARM public test machine is interesting because it supports only 32-bit addition, subtraction, and multiply in hardware. The other operations are supported by either compiler or library emulation. For the hardware supported operations, we derived the operation latency in the standard way. That is, we measure the execution time of a large number of executions of the instruction being characterized, with each instruction depending on the results of the previous instruction. All truth values were determined using random input values.

All floating point operations and all divide instructions are emulated on the ARM system using library calls. We measured truth values for these operations in the same way that we did for native instructions, except that we inserted call instructions instead of native arithmetic instructions. However, because the operations are implemented as software algorithms, we observe slight variability in most of the results. This effect is most pronounced in 64-bit integer divide. For this operation, the execution time varies significantly depending on the operands used. For operands with a small number of significant bits, or for operands that give a result close to one, we observe shorter execution times than for other operands. For this divide operation, we choose the highest latency result over a number of random inputs as the truth value.

For 64-bit addition, subtraction, and multiplication operations on the ARM system, the compiler replaces the 64-bit instruction with an appropriate sequence of 32-bit instructions. For these operations, we can get different results depending on the benchmark, compiler version and compiler optimization flags used. To resolve this issue we report the lowest latency observed using the default compiler. This represents the expected compiler instruction selection.

In general we found that the truth values are not as clearly defined for emulated instructions as they are for the native hardware instructions.

4.6 Parallel Integer, Floating Point and Memory Contexts Values

We implemented a set of benchmarks to determine the number of effective parallel contexts available to threads performing work of different types. Each benchmark consists of a compute kernel that is intensive in either integer, floating point, or memory operations. To determine the number of parallel contexts for one of these kinds of kernel, the benchmark spawned increasing numbers of threads that execute the appropriate compute kernel in parallel with the other threads. When a separate context is not available for some thread, the amount of time it takes for all threads to complete the compute kernel shows a marked increase. The number of parallel contexts for a given kind of kernel is defined as the total number of threads that were able to execute in parallel without observing a drop in performance.

4.6.1 Live 32 and 64-bit Integer and Floating Point Register Values

For each system, we examined the processor architecture along with its documentation to determine the number of live registers that are available to four types of computations: 32-bit integer, 64-bit integer, 32-bit floating point, and 64-bit double precision floating point. The number of live registers was determined by counting the number of general purpose registers of the applicable type that are available, and removing any registers that are typically used or reserved for use by the compiler.

For example, Dash has 16 general purpose 64-bit integer registers available at any given time. However two of these registers, named **%rsp** and **%rbp** for the stack pointer and the base pointer respectively, are generally used by the compiler to store information relating to the program's call stack and thus are not available for computation.

4.6.2 NUMA Values

We measure NUMA values with Multi-MAPS. If we were to extend the measurement of memory bandwidth shown **Figure 5** to larger memory sizes we would see some further plateaus on Dash as shown in **Figure 9.**

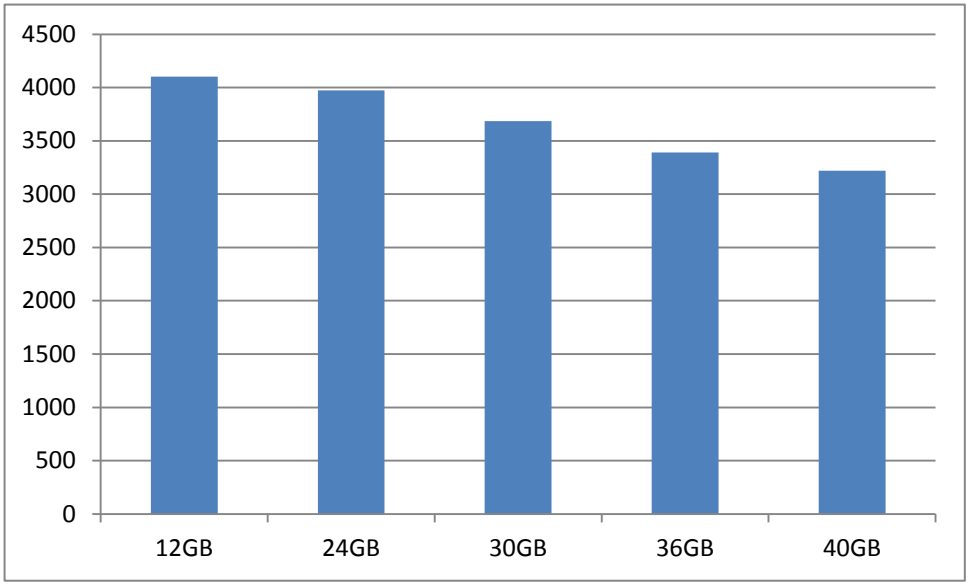

Figure 9: Performance of MAPS stride-1 in MB/s as function of size on Dash
Each socket has 24GB of memory; note clear NUMA drop going out beyond 24GB (and minor drops before that)

For more details about the individual scores for each T1 team see Appendix C.

4.7 Results of Using PIR on Challenge Problems

We used the PIR toolset to characterize the properties of the UHPC challenge applications. Most of these will be familiar to workload characterization experts– but some reflect unique capabilities of the tools.

Runtime and Scaling behavior with problem size, we compiled serial versions of the challenge applications for a single core of a hex-core Xeon 5660, running at 2.8GHz with 12MB L3 cache and a 6.4 GT/s QPI bus. Memory is 24GB DDR3-1333 RAM (six DIMMS, 4 GB / DIMM). The challenge problems were compiled with gcc 4.4.3 using the –O3 optimization level (aggressive optimization) but

without including more aggressive optimizations, such as –ffastmath (an even more aggressive level of optimization). Except where noted, runtimes reported are elapsed wall clock time.

Compute Idioms are based on PIR [18] and formally defined in Appendix D. PIR is an automated idiom classifier which analyzes the array reference patterns within a loop nest and specially augmented via the MAACE retargeted funding. This system has been shown to recognize memory access idioms with high accuracy.

Overall and Temporal Data, Operation, and Instruction Types, we used PEBIL [19], a high performance static binary rewriting tool which allows the insertion of instrumentation for basic blocks, which when combined with static analysis of the basic blocks, yields accurate counts for all of these characteristics with modest runtime overhead. To create the temporal counts, we extended PEBIL with a periodic sampling infrastructure.

Finally, we explored the **Memory Behavior** of the five UHPC challenge applications using SESC [20], a cache simulator, and varying L2 cache (last level cache on chip) sizes. SESC is a cycle accurate microprocessor architectural simulator with many capabilities; we use it simply to investigate the memory reference behavior and resulting energy implications for the challenge applications on a range of L2 cache sizes.

4.8 Basic Challenge Problem Characteristics

We explore basic application characteristics and scaling, capturing the application runtime as key problem size parameter is varied.

For the Streaming sensor challenge problem, the runtimes include image formation (backprojection), registration (affine and thin plate spline) and change detection (CCD and CFAR). We reconstruct two images to facilitate registration and change detection. Thus, the image formation algorithm is performed twice while the registration and change detection algorithms are each performed once.

The sensor problem uses simulated data from a data generator. Larger images require more data for sampling reasons, so we scale the problem size by repeatedly doubling the per-side image dimension (quadrupling the number of pixels) and doubling the number of pulses. Backprojection runtime increases with the number of image pixels times the number of pulses (each pulse is backprojected onto each pixel). The image sizes vary from 256x256 to 4096x4096. For these larger data sizes, runs for a single image is many hours on a single core.

Table 6: Basic Challenge Problem

SENSOR		
Image Size (Num Pixels)	Num Pixels * Num Pulses	Run-time (s)
131,072	35,389,440	15
2,097,152	2,264,924,160	930
33,554,432	144,955,146,240	61,468
CHESS		
Ply Level		Run-time (s)
4		0.9
5		29
6		831
7		28,133
SHOCK		
Domain size		Run-time (s)
1,000		0.6
125,000		235
729,000		2,144
2,197,000		6,756
4,913,000		16,108
8,000,000		27,900
GRAPH		
Scale		Run-time(s)
12		0.0007
15		0.007
18		0.075
21		1.0
24		12.6

The Chess challenge problem runtime grows exponentially with ply depth. The ply level corresponds to the decision tree depth, so increasing the ply level by one generates a tree that looks one more move into the future. The reported run-time is the average of three runs, and reported based on internal application calls to ftime(). For even modest plys of depth seven, the evaluations can be many hours on a single core. In the Shock hydrodynamics challenge problem, the modeling is performed over a domain with a cube of equally-sized elements, so increasing one edge size by a factor of two increases the domain size by a factor of eight. Data is included for the number of edge elements ranging from 10 to 200, and it can be seen clearly that the runtime grows rapidly with domain size. The Graph application grows exponentially in runtime with the scale factor. Molecular Dynamics problem is not included because the current set of input systems provided for MD span only a small range of problem sizes unsuitable for meaningful scaling.

4.9 Idioms

To characterize the memory use idioms in the UHPC challenge applications, we ran the PIR tool which automatically classifies loop nests as *compute idioms* on all five applications. The compute idioms are five defined formally in [17] (repeated below in Appendix D) and are 1) STREAM a sequential data access pattern, 2) Transpose a reordering of data row major to column major, 3) Stencil a regular access pattern on the loop induction variable, 4 and 5) Gather/Scatter data to arbitrary (random) memory locations.

Table 7: Compute Idiom Usage Percentage in the UHPC Challenge Problems
Automatically classified – see Appendix D

Application	Type of idiom				
	Reduction	Transpose	Stream	Stencil	Gather/Scatter
Chess	14.3	47.6	9.5	0.0	28.6
Graph	4.6	42.6	11.1	0.0	24.1
Molecular Dynamics	6.3	21.4	7.8	3.5	17.4
Streaming Sensor	4.2	32.6	15.0	0.0	21.8
Shock Hydrodynamics	0.0	31.3	6.1	1.2	17.1

The idiom usage data is normalized by the number of statements in loops such that at least one statement in the loop fits one of these idioms (the missing percentage from 100 is unclassified statements in such loops) While the fraction of all such loops (loops with at least one recognized idiom) varies significantly (from as high as 80% of the loops to as low as 25%), the classifications nonetheless show the variety and weights of compute idioms extant in the program. In particular, the idiom classifier shows that all of the

challenge problems have complex, irregular memory access patters (gather, scatter, reduction) as well as ordering challenges (transpose), suggesting that supporting these access patterns well is critical for UHPC compute engines.

4.10 Data Types and Operations

Using PEBIL, we instrumented all five challenge applications, counting the instruction types and recording operations. These statistics we collected into a number of key categories to distinguish their needs (see Appendix A for precise binning definitions). In general we want to know what classes of operations, arithmetic, memory, conditional, etc. are exercised by each benchmark. Next we discuss these results per benchmark as well as their dominant idioms.

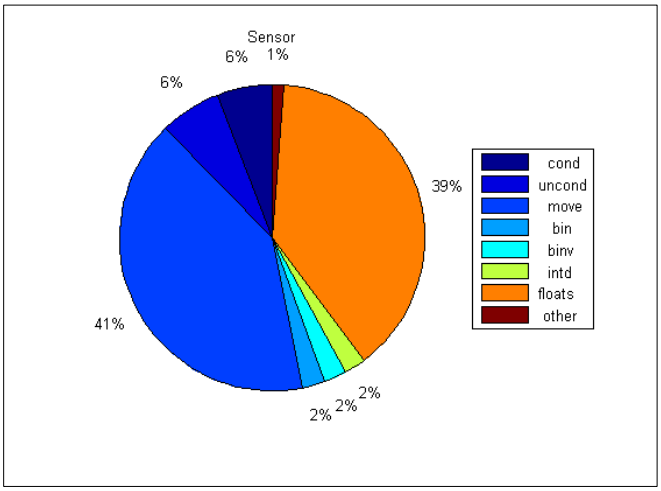

Figure 10: Streaming Sensor

The streaming sensor application is dominated by control flow operations, short integer operations, and data movement (streaming) and transpose. While this application uses single precision floating point operations (the sensor data), in other sensor problems this data may be represented by short fixed precision operations.

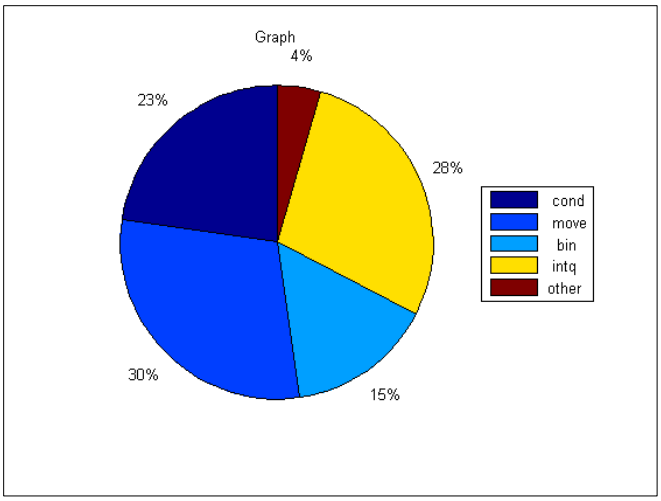

Figure 11: Graph

The graph application is control flow and data movement intensive. The major computational operations are integer and binary operations. Large datatypes are used (quadwords) and notably no floating point operations. Its data access patterns are dominantly transpose or random.

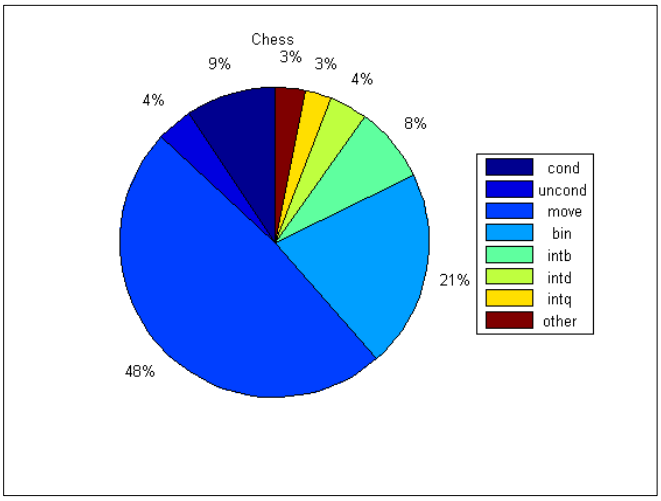

Figure 12: Chess

The chess challenge problem is an exemplar of search and decision support applications. It is heavy on integer, control flow, and data movement oriented and again essentially no floating point. Its data access patterns are highly random and transpose.

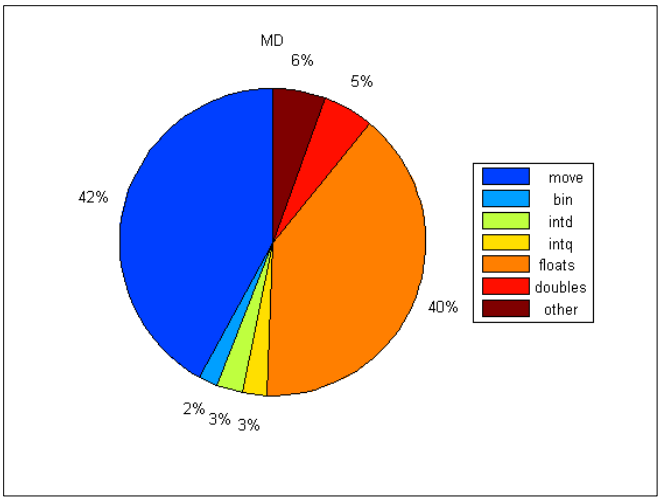

Figure 13: Molecular Dynamics

The molecular dynamics challenge problem is heavily floating point intensive, making regular use of single and double precision floating point operations. Of the remaining instructions, the data movement overhead is notable, perhaps reflecting the small register file sizes in the x86 architecture.

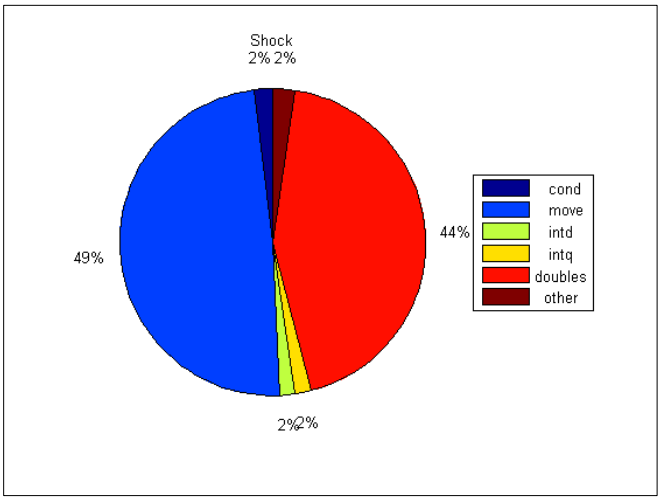

Figure 14: Shock Hydrodynamics

The shock hydrodynamics code is a highly-optimized scientific code with large numbers of floating point operations – single and double – dominating the instruction counts. Interestingly, little or no use is made of the vector floating point operations, suggesting that the compiler (or libraries) didn't make the additional investment to exploit this next level of performance on the x86 platform.

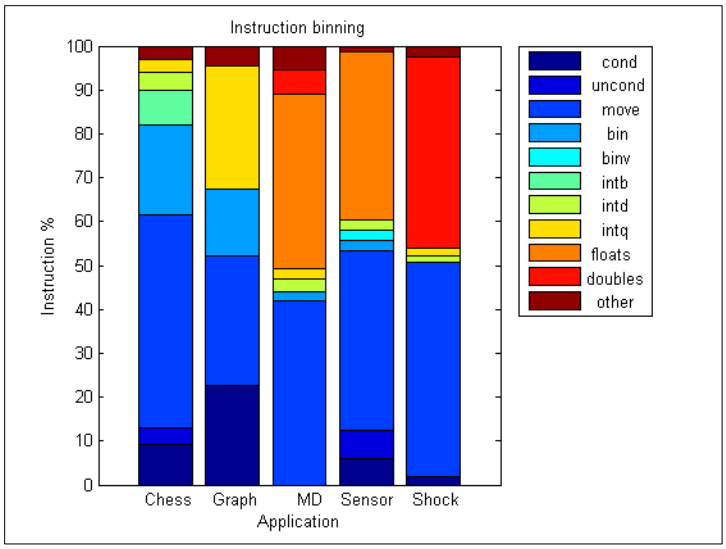

Figure 15: Overall Summary

The summary bar graph shows the clear variation across challenge applications. Chess and graph make heavy use of integer operations and control flow, and little use of floating point. Across the others, the usage of floating point varies. In all cases, heavy use of move operations reflects the paucity of registers. As before, none of these applications are dominated by simple sequential access patterns.

4.11 Data Types and Operations with Temporal Binning

We expect temporal binning data to illuminate even greater phase variation within applications.

4.12 Memory behavior and Energy per Operation

The DARPA Exascale hardware report [3] includes a "strawman" design that depends critically on high levels of data locality to approach the ambitious energy-efficiency targets. Specifically, the strawman assumes a 50:1 ratio floating point operations and off-chip DRAM access, and a 12.5:1 ratio between program memory references and off-chip DRAM references. The data locality targets are critical for a UHPC to achieve the desired energy efficiency. In this vein, we study the data locality properties of the UHPC benchmarks to explore their need for working sets and data movement.

For this evaluation, we use the SESC simulator [19]. We model a uni-processor system consisting of a 3GHz processor with a two-level cache hierarchy and a DDR3-based main-memory system. The processor is a 3-issue out of order core. The L2 cache has 64-byte lines and 8-way associativity. We vary its size from 32Kbytes to 16 Mbytes. The L1 cache has 64-byte lines and 16-way associativity. To model energy and power in our simulations, we use data from the International Technology Roadmap for Semiconductors (ITRS) [30] and CACTI [31] corrected for 32nm technology. These details are included for reproducibility but the thrust of our investigation is to explore *qualitatively* the locality properties of the benchmarks (the trends are more important than the simulator details).

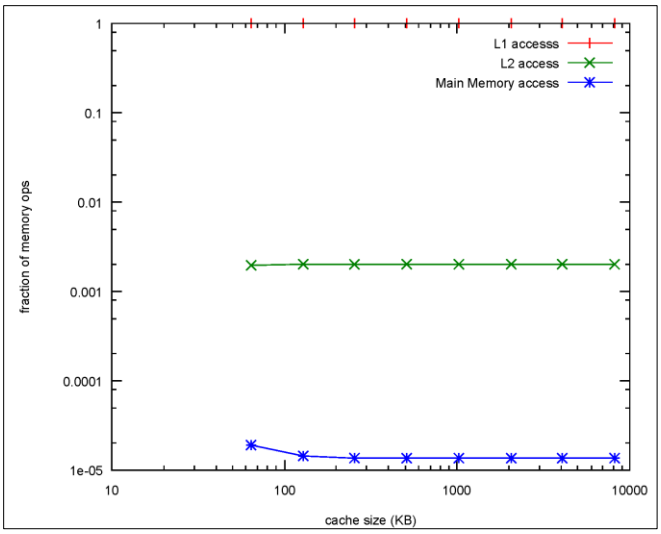

Figure 16: Streaming Sensor (SAR Backprojection)

The streaming sensor application is a model of cacheability, with extremely high hit rates in L1 and L2, even with modest size caches.

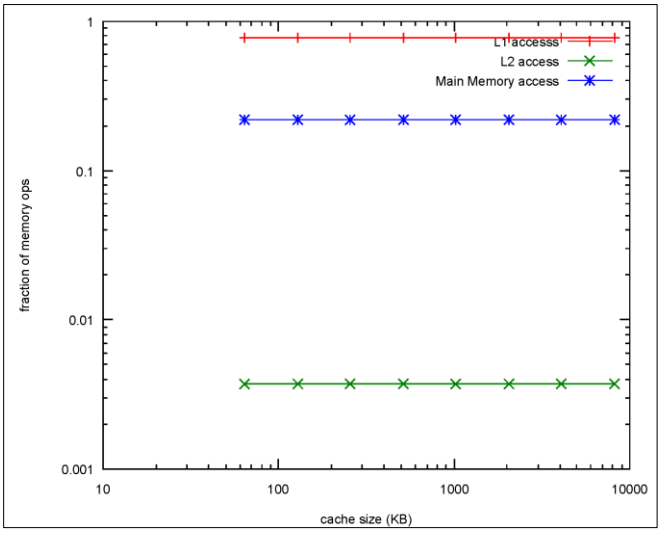

Figure 17: Graph

A full 25% of the graph problem's accesses go to (DRAM), independent of the size of the L2 caches. In effect, the working set is captured by a small L2, and a large number of references go all the way to DRAM.

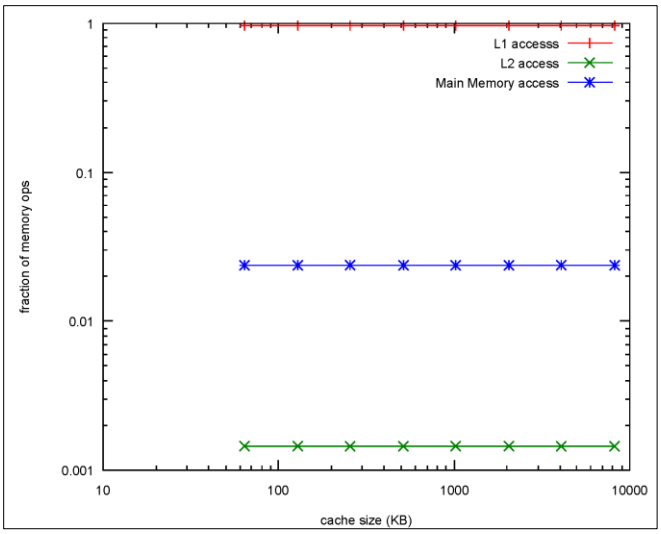

Figure 18: Chess

The chess problem can fit almost completely in a modest sized L2 cache, but still retains a DRAM access rate of nearly 2%, nearly 10-times the other cacheable challenge problems.

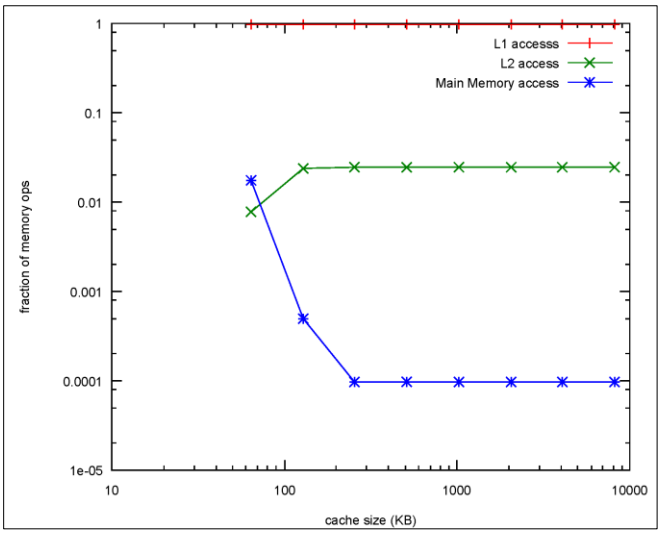

Figure 19: Shock Hydrodynamics (Lulesh)

Lulesh needs a larger L2 Cache than chess but can be almost completely fit into a medium sized L2.

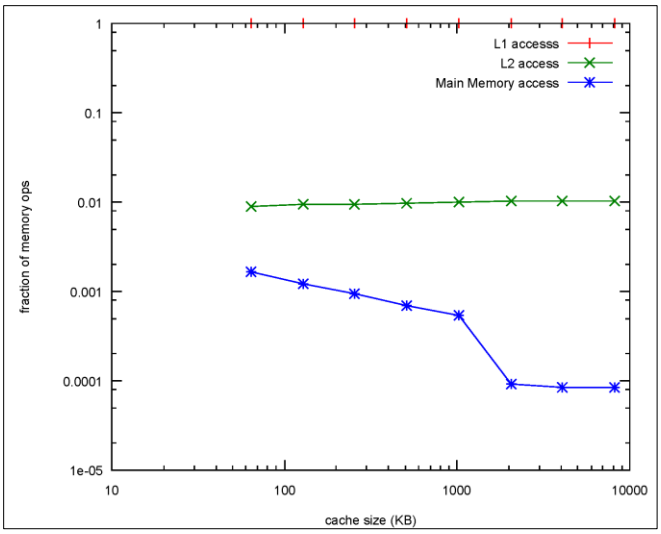

Figure 20: Molecular Dynamics

MD is similar to Lulesh, with a large working set that can be captured by a 2MB L2 cache.

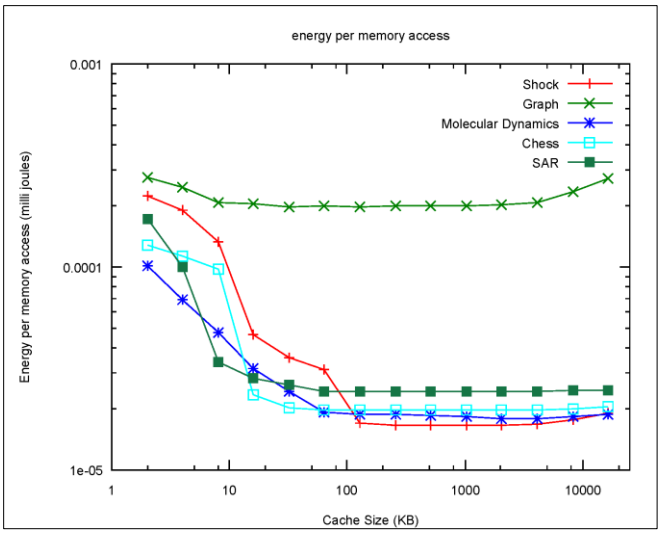

Figure 21: Memory System Energy Usage

Our SESC simulator experiments show that the majority of the UHPC challenge problems exhibit high-levels of data locality. With modest sized L2 caches, most of the challenge problems (Chess, MD, SAR, and Lulesh/Shock Hydro) can have external access ratios close to the 12.5:1 targets posited in the DARPA Exascale report. It's worth noting the ratios are inflated (positively) in these simulations by the ratio of the L2 block size to access size. The Graph problem however, doesn't exhibit sufficient data

locality. With a 20% L2 miss rate, its memory system energy usage will be challenging for exascale systems.

4.13 Related Workload Characterization

Many application workload models have been developed, including the SPEC benchmarks [21], the HPC Challenge (HPCC) benchmark suite [22], the Berkeley Motifs [23], and PARSEC [24]. These respectively focus on workstation/PC performance, supercomputers, parallel computing, and future microprocessors (visual and data-intensive). The HPCC benchmark suite shares some of the same goals. The HPCC test set has some similarities to our idiom set, but idioms are all on the same low level and more general in application codes.

4.14 The UHPC Benchmarks

The challenge problems represent a unique mix – that span the scales of data center, workgroup, and embedded. These three distinct tiers are only recently coming to be understood as closely tied and related – by coupled applications, increasing needs for interoperability, and shared challenges in energy efficiency. The UHPC benchmarks are distinct in their breadth of focus, ranging far beyond the traditional desktop/workstation workloads such as SPEC, and HPC workloads such as HPCC challenge and Berkeley Motifs.

The UHPC challenge diversity is clearly reflected in our characterization results – the workloads are not floating point dominated, making heavy use of fixed point data types. They also make heavy use of control flow and data movement, and exhibit complex, irregular data access patterns. The UHPC challenge problems are indeed quite a diverse and challenging set of computational structures.

5.0 Conclusions

5.1 General Issues

5.1.1 Configuration Information

The final versions of the T1 teams' characterization tools do not yet make use of the configuration XML file defined during Phase 1. PACE's tools make use of a custom configuration file that uses default values that happen to be appropriate for all of our public and private machines. AESOP forgoes configuration information entirely, and relies upon standard environment flags and the GNU Autoconf tool-chain to determine configuration parameters.

In discussion with both teams, we learned that neither wanted to parse the XML file directly. To alleviate this problem, we developed a conversion tool that converts the XML configuration files into a format that fits into both team's existing solutions: a compatible flat file format for PACE, and a set of standard environment variables for AESOP. However, consensus concerning the exact behavior of this tool was never reached in time for either team to integrate this tool into their final Phase 1 release.

Had there been a Phase 2 the plan was to rectify this situation, either by helping them integrate our conversion tool or convincing them to directly use the configuration XML files as was initially intended.

5.1.2 Semantics, Operation Characteristics, and the C Compiler

One issue encountered during Phase 1 was how to best define the semantics of various characteristics. While the meaning for most of the characteristics was resolved during the phase, there are a few remaining characteristics that are difficult to precisely define in an exact, cross-platform manner. The operation characteristics such as simultaneous operations, operation latency, and operation throughput cannot be defined by a single value, largely due to the interaction with the target C compiler. This arises from the fact that the architecture tends to behave very differently given seemingly similar, but slightly different instruction sequences. Furthermore, trying to generate a specific instruction sequence given the opaque C compiler further complicates the matter. We discuss specific instances of this issue in the discussion of PACE's simultaneous operations in Section 10.2.10.

5.1.3 Run-to-run Variations in the Characterization Tools

We noticed significant variation in the values reported for some characteristics. Both characterization tools exhibit inter-run variation on all machines, but the set of characteristics that vary differs from team to team and machine to machine. To ensure the integrity of our evaluation, our official score is based on running each team's characterization tool one time.

In PACE's case, the characteristics that vary on most machines are cache size (except for L1) and floating point simultaneous operations. Other characteristics that vary significantly, though only on a few machines, are L2 cache latency, L2 cache line size, integer simultaneous operations,

operation latencies, and parallel contexts. As an example, **Table 8** lists the variation in PACE's values for SDR. The variations are calculated using the following formula: (maximum – minimum) / minimum.

Table 8: Variation in PACE's Values for SDR

Characteristic	Variation
L2 cache line size	100.00%
L2 cache size	25.00%
flt32 add simultaneous operations	300.00%
flt32 sub simultaneous operations	300.00%
flt32 mul simultaneous operations	300.00%
flt32 div simultaneous operations	66.67%
flt64 add simultaneous operations	200.00%
flt64 sub simultaneous operations	300.00%
flt64 mul simultaneous operations	300.00%
flt64 div simultaneous operations	250.00%

In AESOP's case, the one characteristic that varies on most machines is the L2 cache size. Other characteristics that vary significantly, though only on a few machines, include L1 cache size, cache line sizes and associativities, operation throughputs, parallel contexts, and NUMA node size and count. As an example, **Table 9** lists variation in AESOP's values for PS3-PPE. Again, the variations are calculated using the following formula: (maximum – minimum) / minimum.

Table 9: Variation in AESOP's Values for PS3-PPE

Characteristic	Variation
L1 cache size	700.00%
L2 cache line size	700.00%
L2 cache size	300.00%
L2 cache associativity	100.00%
flt32 add throughput	10.79%
flt32 sub throughput	10.02%
flt64 add throughput	10.79%
flt64 sub throughput	10.79%

We discuss the reasons for the variations on a case-by-case basis in Section 10.2.

5.1.4 Rounding Issues for Operation Latency

In the results for operation latency, there is a difference of 1 between our truth values and the measured values from the AESOP and PACE characterization tools. We believe this is due to rounding. For example, in int64 add and subtract, PACE reported 1.2 while our truth value was 1.79. When these two values are rounded, the results are 1 and 2, respectively. Our Wiebull function takes this discrepancy into account when grading.

5.2 Detailed Analysis of Characteristics with Significant Differences

In the following sections we give a detailed analysis of the characterizations with significant differences between our truth values and the T1 measured values.

5.2.1 AESOP Cache Sizes on SDR and PS3-PPE

Incorrect Cache Hierarchy and Variation

For the private machine SDR, the AESOP characterization tool sometimes reports incorrect sizes for the L1 cache. When this error occurs, 8,096 bytes is reported as the L1 cache size and what would otherwise be a correct L1 measurement is incorrectly reported as the L2 cache size. It is simply by chance that this problem occurred during the run that was scored. We also find that the

size of the L2 cache size is frequently overestimated as 6,291,456 bytes whereas we found an effective L2 cache size of 4,333,568 bytes.
We observed a similar behavior on the PS3-PPE private machine. For this machine, the AESOP characterization tool sometimes misses the L1 level of cache and incorrectly reports the measured L2 cache size as the L1 cache size. For the execution of the tool that was scored for this report, this problem did not occur.

The AESOP cache benchmark measures memory access times over arrays of varying sizes. These timing results are then automatically analyzed to determine the cache sizes. Our experience suggests that this process is sensitive to noise for L1 caches sizes, resulting in missed or extra cache levels. The frequent overestimate of L2 cache size for SDR is simply the result of a poor choice by the analysis routines. When we manually analyzed the detailed timing results from the benchmark, we found that a better answer should have been found much closer to our truth value.

Rounding to powers of 2

For the private machine PS3-PPE, the AESOP tool reports a L2 cache size of 524,228 bytes whereas we report a truth value of 360,448 bytes. When we investigate the detailed results of the benchmark, we can see that, for cache sizes below 2,097,152 bytes, the AESOP tool is only testing for cache sizes that are 2^N * 1024 bytes. At this granularity, the only other reasonable choice the tool could have made is 262,144 bytes. Neither of these numbers capture the *effective* cache size, or the cache size seen by software. It is our opinion that the granularity used by the AESOP tool is too large to correctly measure the effective cache size on the PS3-PPE system.

5.2.2 AESOP L1 Cache Line Size on SDR

As discussed above, AESOP's cache characterization tool exhibits run-to-run variation. On SDR, AESOP's tool reports a range of values for the L1 cache line size varying from 64 to 1024 bytes. The value of 64 bytes matches our computed truth value. While we scored AESOP based on a single run of their characterization tool, it should be noted that AESOP occasionally produces the correct value for this characteristic.

5.2.3 AESOP L2 Cache Associativity on SDR and PS3-PPE

For all levels of cache, we determined the cache associativity truth values based on the published machine specifications or equivalent. On x86 machine such as DASH, SDR, and Triton, we used the associativity values reported by the CPUID instruction. For the other machines, we used the published specifications. Our position is that the semantics of the cache associativity metric discussed with the T1 teams is compatible with the published values. While AESOP's values differ from these published values, it is unclear whether these differences are due to a methodology or implementation error in AESOP's approach, or a difference in semantics. Had there been a Phase 2 the plan was to work with both PACE and AESOP in Phase 2 to more clearly define the semantics of the cache associativity characteristic.

5.2.4 AESOP Float and Memory Contexts Values

The truth values differed from AESOP's measured values for float and memory contexts on DASH, Triton, SDR, and Batcave. For all machines, our reported truth values mirror the number

of available kernel threads, while the values reported by the AESOP characterization tool are lower than our own. Lower values seem to indicate that the benchmarks employed by AESOP results in greater conflicts between threads, which would cause a performance drop to occur with a smaller number of running threads. This reality implies that the computed metric is sensitive to the behavior of the compute kernel employed (*e.g.* the difference between two memory-intensive threads *vs.* one memory-intensive and one compute-intensive thread), and is therefore another semantic issue that would have been addressed to be further refined in Phase 2.

5.2.5 AESOP Divide Latency and Throughput on ARM and PS3-PPE

We observed that divide operations can have operand dependent latency. The simplest example of this is a divide by zero or one where the divide algorithm can return an answer after doing very little work. This is especially true for divide operations that must be emulated in software by the compiler or an external library. For example, our ARM public machine does not support 64-bit divide in hardware. For this operation we can obtain results that vary from 125 cycles to 450 cycles based completely on the inputs. For this machine, the AESOP characterization tool gives integer divide latency and throughput numbers that are similar to the mean value returned by our benchmark with varying inputs. The differences between AESOP's values and our truth values for integer divide on PS3-PPE are likely due to the same reason. According to IBM Cell Processor documents, integer divide has variable latency.

5.2.6 AESOP NUMA Values

It does not appear that AESOP's characterization tool ran NUMA tests far enough out into main memory to detect NUMA behaviors in DRAM on multisocket systems such as Dash, which is why their numbers disagree with ours. See Figure above for an example of what they would have detected if they had done so. In fact, systems such as Dash (a public system) that are based on the Intel Nehalem CPU are well documented to have NUMA characteristics between sockets. Interestingly, AESOP claims that they do not need to characterize L3 but do need to characterize DRAM NUMA. However, one can see that one cannot detect where the latter starts unless one knows where the former begins. We think that the surprisingly large NUMA numbers on Batcave for example may actually be cache effects rather than DRAM NUMA effects – *i.e.* they may think there is a local memory when in fact it is a large local SRAM cache.

5.2.7 PACE L2 Cache Sizes on SDR and Triton, and L2 Cache Line Size on SDR

PACE's cache characterization tool exhibits run-to-run variation. On SDR, PACE's tool reported a range of values for the L2 cache size varying from 4194304 to 5242880 bytes. The value of 4194304 bytes is reasonably close to our computed truth value. On Triton, PACE's tool reported a range of values from 196608 to 262144 bytes. The value of 262144 bytes matches our truth value.

This run-to-run variation also affects the L2 line size measurements. On SDR, PACE's tool reported a range of values for the L2 cache line size varying from 64 to 128 bytes. The value of 64 bytes matches our truth value. Furthermore, the L2 cache and line size measurements appear to be linearly related, thus explaining why the run-to-run variance affected both characteristics.

While we scored PACE based on a single run of their characterization tool, it should be noted that PACE occasionally produces a reasonably correct value for this characteristic.

We investigated the variation in PACE's cache benchmark and determined that the issue is due to a relaxed methodology rather than being completely incorrect. For example, on SDR, PACE's tool always reports 4MB or 5MB as the L2 cache size, while our truth value is close to 4MB. However, the 5MB value is not inherently wrong. Our custom cache latency benchmark can also be used to determine cache sizes, and will report either 4MB or 5MB depending on the access pattern invoked during bbbbbbhe test. On the other hand, our Multi-MAPS benchmark, which determines our cache truth values, consistently reports values around 4MB. PACE's benchmark uses an access pattern that can trigger either behavior depending on the state of the system, yielding the two different results. Both 4MB and 5MB can be considered correct values depending on how the cache size characteristic is defined.

We planned to work with both teams in Phase 2 had there been a Phase 2to further refine the semantics of the cache size characteristic and converge to a common access pattern methodology that yields consistent results between runs.

5.2.8 PACE Level 2 TLB on Triton

The PACE characterization tool erroneously does not generate values for the Level 2 TLB. Earlier versions of the PACE tool correctly generated values for Level 2 TLB. For example, the PACE tool released on July 5th reported a Level 2 TLB capacity of 2,097,152 bytes (the actual capacity is 4,194,304 bytes) and a Level 2 TLB page size of 4,096 bytes (which is correct). We noted a significant code modification in the file TLBTest.c between the July 5th and final versions that may be linked to this error.

5.2.9 PACE L3 Cache Latency on DASH, Triton, Batcave

When measuring cache read latency, there are three issues that can corrupt the result:

1. Data may hit in lower level of cache, thereby lowering the perceived latency.
2. The hardware pre-fetcher can preemptively load data into lower cache levels, again lowering the perceived latency.
3. The cache under test (usually L3) may hold more data than the TLB can hold references for, so TLB misses can increase the perceived latency.

Our cache latency benchmark is designed to address these challenges. First, our benchmark never touches the same cache line more than once, so we avoid the issue of a future access hitting in a lower cache level. Second, our benchmark uses a random access pattern to prevent the hardware prefetcher from correctly predicting future accesses. Finally, our benchmark breaks larger caches into smaller sub-blocks that are tested individually. These sub-blocks are chosen to guarantee that they fit into the TLB, so we only cause a fixed number of initial cold misses. These cold misses are then amortized across the entire sub-block test, resulting in very little perturbation to the computed value. Given this design, we have high confidence in our computed cache latency values. Our L1 and L2 latency values largely agree with PACE and AESOP on all public and

private machines, however our L3 latency values differ from PACE on all machines that have an L3 cache: DASH, Batcave, and Triton.

PACE's characterization tool runs two different cache benchmarks which compute different values. At the end of the characterization sequence, the tool does a number of checks to reconcile the values produced by the two benchmarks. For the latency value, the reconciliation phase simply chooses the smaller of the computed L3 latency values. While we have not inspected the two benchmarks to determine their differences, we have noticed the benchmark that produces the higher value more closely mirrors our reported truth values. If we change our benchmark such that individual cache lines are hit more than once, our benchmark produces faster latency values that are similar to PACE's reported (smaller) values, so we hypothesize that the results reported by PACE are influenced by issue #1 discussed above (references hitting in lower cache layers).

5.2.10 PACE Simultaneous Operations on All Machines

We uncovered four problems regarding PACE's measurement of simultaneous operations. First, there is a coding error that causes their benchmark for floating-point simultaneous operations to run with the wrong number of iterations on SDR, Triton, PS3-PPE, and DASH. Below, we give a detailed discussion of this issue, including how PACE's scores and quality metrics change if this bug is fixed. The second issue regards the variability in the code generated by the C compiler, causing the measured values to be highly sensitive to small changes in the C code used for the benchmark. We discuss the nature of this variability below. Third, there is a problem in how PACE analyzes the measured values to decide on the final value for simultaneous operations, which we also discuss in more detail below. Finally, we discuss how variations in the run environment cause the PACE characterization tool to report values that vary from run to run.

Bug in PACE Simultaneous Operations Measurement Tool

The PACE characterization tool creates an array of function pointers, where each function handles a type-operation pair. The first parameter is a value of the given type (int32, int64, flt32, flt64) and the second parameter is the number of iterations. However, there is a non-portable implicit cast (the addresses of functions generated by the code on line 317 of generate_issue_slot_code_fixed_length.c assigned to function pointers defined in line 21 of calls_membench.h) that is applied to these function pointers, causing the number of iterations to be corrupted. As a result, on the x86 and PowerPC machines, too few iterations are used to measure simultaneous operations (just 3 instead of thousands), so the timing function does not have the needed resolution. The PACE tool reports anywhere from 2 to 8 simultaneous operations on SDR and 1 to 10 simultaneous operations on PS3-PPE.

We fixed the PACE characterization tool by changing the type of the formal parameter list of all functions to (int, int). (The first argument is not used within the body of any of the functions, so changing its type does not change the behavior of the functions.) We ran the PACE characterization tool with this fix on all the systems **Table 10 and Table 11** compare the floating-point simultaneous operations results for private and public systems with and without the fix.

Table 10: Floating-point simultaneous operations results for all the private machines using latest revision of PACE characterization tool with and without the fix

	PS3-PPE			SDR			Triton		
Characteristic	Truth	PACE	PACE with Bug Fix	Truth	PACE	PACE with Bug Fix	Truth	PACE	PACE with Bug Fix
FLT32 ADD	10	3	15	3	2	3	3	2	3
FLT32 SUB	10	2	15	3	3	3	3	3	3
FLT32 MUL	10	3	15	4	3	4	4	3	4
FLT32 DIV	1	1	1	2	3	1	1	3	1
FLT64 ADD	10	2	15	3	5	3	3	2	3
FLT64 SUB	10	3	15	3	5	3	3	2	3
FLT64 MUL	10	3	15	g 5	3	5	5	2	5
FLT64 DIV	1	1	1	2	5	1	1	1	1

Table 11: **Floating-point simultaneous operations results for all the public machines using latest revision of PACE characterization tool with and without the fix**

	ARM			Batcave			Dash		
Characteristic	Truth	PACE	PACE with Bug Fix	Truth	PACE	PACE with Bug Fix	Truth	PACE	PACE with Bug Fix
FLT32 ADD	1	1	1	8	8	8	3	2	3
FLT32 SUB	1	1	1	8	8	8	3	4	3
FLT32 MUL	1	1	1	8	8	8	4	2	4
FLT32 DIV	1	1	1	1	1	1	1	2	1
FLT64 ADD	1	1	1	8	8	8	3	2	1
FLT64 SUB	1	1	1	8	8	8	3	2	3
FLT64 MUL	1	1	1	8	8	8	5	3	5
FLT64 DIV	1	1	1	1	1	1	1	2	1

Table 12 compares the scores and quality metrics for all the systems with and without the fix. Note that whereas some systems had quality metrics below 75% without the fix, all have quality metric greater than 75% with the bug fix in place.

Table 12: **PACE's scores and quality metrics for all the systems with and without the fix**

	Scores		Quality Metrics	
Machine	Without Fix	With Fix	Without Fix	With Fix
PS3-PPE	93.825%	94.002%	76.167%	76.730%
SDR	92.625%	94.347%	71.917%	83.747%
Triton	93.521%	95.128%	75.272%	87.412%
ARM	97.347%	96.868%	95.254%	93.741%
Batcave	96.765%	96.818%	91.463%	91.670%
Dash	93.134%	94.658%	73.012%	84.844%

Variability in C Code Generation

For simultaneous operations for integers, we were able to produce two different C benchmarks that both appeared to measure the same characteristic but had different results. This is because many C compilers choose different instructions for a C language "add" operation depending on the context. These different instructions have different characteristics on the target machine, with neither of the variants being unambiguously better or worse – just different.

For example, on the IA-32 architecture, if register %edx contains the value x, then the instruction leal 7(%edx,%edx,4),%eax will set the %eax register to "5x+7" (by computing x + 4*x + 7). This same expression can be calculated by a series of add and shl instructions. On some implementations of the IA-32 ISA, a processor can issue a single lea instruction while it can issue 4 add instructions; this would result in a different number of operations in flight. Which operation is chosen is dependent on the underlying compiler.

One of our benchmarks matched the results reported by PACE, while another benchmark matched what appears to be the theoretical hardware maximum for basic integer operations. For Phase 1, we grade PACE's characterization tool based on the second value, thus slightly lowering its score.

In the end, the value that matters the most is based on the type of code the T1 compilers eventually generate. If the compilers generate code that is more similar to the first benchmark, than PACE's values are correct. Otherwise, they are not. This would have been a primary focus issue for MAACE in Phase 2 had there been a Phase 2.

Error in the Analysis of Simultaneous Operations

With the fix, the PACE characterization tool produces values for simultaneous operations that mostly match our values. Of the differences, some, such as int32 and int64 adds on SDR, Dash, and Triton (all x86-64 machines), can be explained by the variability in C code generation issue that we described above. Some other differences, such as flt32 and flt64 adds, subtracts, and multiplies on PS3-PPE, are due to a problem in how the PACE tool analyzes the measured values to decide on the final value for simultaneous operations. When we manually analyze the measured values, we arrive at a final value that matches the value we obtained using our benchmark and manuals. We believe the automated analysis doesn't work for some systems because PACE has fine-tuned it for some systems but not others.

For an example of the flawed automated analysis, consider **Figure** which shows the data generated by the PACE characterization tool with the fix for flt32 add on PS3-PPE. The plot contains 16 data points. The first of these data points reports the time it took to execute a single "stream" of operations on PS3-PPE. As before, a stream of operations is defined as a dependent sequence of operations. The next data point reports the time it took to execute two independent streams, where every operation in the second stream is not dependent on any operation in the first stream. Doubling the work by adding a second stream did not change the overall time to execute all the operations (ignoring noise) on PS3-PPE, which means that the operations in the first and second streams were executed in parallel. The PACE characterization tool introduces further

independent streams (up to 16) and measures the time for each case. It then performs analysis on the data to find the maximum number of streams that can execute in parallel. Once it finds this number, the tool reports it as the value for simultaneous operations.

Looking at the plot, we see that the maximum number of streams that can execute in parallel is clearly 10 for PS3-PPE, yet the PACE tool reports 15 as the value for simultaneous operations.

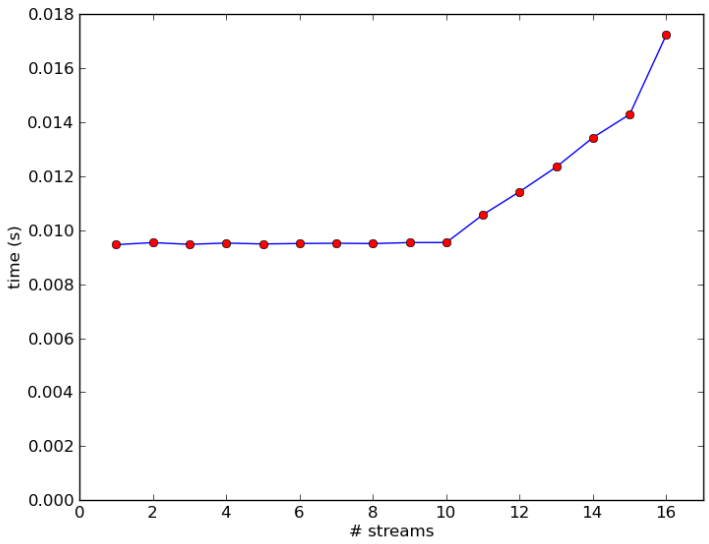

Figure 22: **Data points collected by the simultaneous operations microbenchmark in PACE characterization tool for flt32 add on PS3-PPE**

This occurs because PACE test uses the derivative (delta) of the values to determine when a change has occurred. Their (conservative) test assumes that change occurs at 15 streams; our visual inspection indicates the true value of 10 streams. We believe this is the true value because the PS3-PPE has a 10-stage floating point pipeline, which is what is being measured.

Run-to-run Variations in Simultaneous Operations

PACE's benchmark for finding simultaneous operations exhibits run-to-run variation as a result of variations in the run environment. While the benchmark produces varying results on almost all systems, the ones that are most affected are Dash, SDR, and Triton. The bug described above is the primary source of variation in the values reported by the benchmark for floating-point simultaneous operations on Dash, PS3-PPE, SDR, and Triton. The variation being discussed here, on the other hand, is seen for both integer and floating-point simultaneous operations on all machines, even with the fix. The deviation from the correct value is in some cases large enough to be deemed problematic if we accept the results of any single run of the benchmark. For example, on Dash, the benchmark reports anywhere from 1 to 5 for int32 multiply. The correct value for int32 multiply on Dash is 3. However, in all cases, the probability that the benchmark produces an incorrect value is very low. For example, the probability that the benchmark reports

an incorrect value for int32 multiply on Dash is 7%. To produce the correct value with a greater probability, we suggest that PACE run their benchmark multiple times and compute the mode on the all the results.

5.2.11 PACE Operation Latency on ARM and PS3-PPE

The PACE characterization tool uses very little compiler optimization by default. This causes differences between our truth values and the PACE values for 64-bit integer divide and multiply on the ARM public machine. This is because these instructions must be emulated by the compiler using a sequence of 32-bit operations. For our own benchmark, the compiler optimization flags used by PACE result in poor instruction selection and inefficient code for the divide operation. We believe the same thing may be happening to the PACE benchmark.

For the private machine PS3-PPE, the differences between the PACE values and the truth values for addition and subtraction are likely due to run to run variation or rounding differences. For integer divide latency on PS3-PPE, PACE reports a value significantly higher than our truth. According to the IBM Cell Processor documents, divide is a variable latency instruction. As a result, we believe this difference is caused by different input values to the divide benchmark.

5.2.12 PACE Memory Contexts Values on PS3-PPE

While the PS3-PPE is a 2-way simultaneous multi-threaded machine, all parallel context benchmarks for all three teams suggest that the system only has one effective parallel context. The only exception is PACE's parallel memory context value for PS3-PPE. PACE's characterization tool reports 2 for this metric, while our truth value is 1. We investigated this issue and believe the value reported by PACE to be incorrect.

PACE's benchmark spawns threads that walk through varying sized arrays, starting with 8 KB. Each iteration measures the execution time and computes a thread throughput (number of threads / time). The benchmark continues to iterate and increase the number of threads until it observes a drop in throughput that surpasses a defined threshold. **Figure 23** shows the debug output produced by PACE's memory parallel context benchmark.

```
Data size 8 KB
Num threads 1: Throughput 0.254118, Time 3.935177
Num threads 2: Throughput 0.475392, Time 4.207050
Num threads 3: Throughput 0.368360, Time 8.144212
Data size 32 KB
Num threads 1: Throughput 0.251557, Time 3.975246
Num threads 2: Throughput 0.171968, Time 11.630037
```

Figure 23: Debug output from PACE's memory parallel context benchmark on PS3-PPE

The output shows the time and throughput for each iteration. Based on throughput, PACE's tool should determine that there are 2 parallel contexts for the 8 KB test, but only 1 parallel context

based on the 32 KB test. However, the characterization tool always reports 2. In addition to this inconsistency, we believe that the 8 KB test is too small to ensure threads conflict with each other in the memory subsystem. The 32 KB is a better indicator and therefore the data from PACE's own benchmark suggests the answer should be 1. For reference, our benchmark uses an array size of 256 KB.

Thus we carried out a thorough evaluation of the ability of the two T1 teams to characterize systems that were unknown to them. The results were impressive. While one team (AESOP) did better than the other qualitatively, the other (PACE) collected more values and when we fixed a relatively minor bug in their code, their Quality improved significantly. In retrospect, the Score formula was too easy; we were trying to incentivize both teams to report a lot of values, which in fact they did. However in some sense the Quality metric is probably a more accurate representation of how well each team did in characterizing the unknown machines. Modulo some bug fixes, both teams achieved 75%+ on the Private machines on Score *and* Quality, so we have no reservations in recommending a PASS for both AESOP and PACE in Phase 1.

5.3 Diversity Motivates 10x10

We have characterized serial versions of challenge problems in a number of dimensions using specially augmented PIR, and shown that they are unusually diverse and therefore challenging to implement with high-performance and energy efficiency. The extant challenge problem diversity highlights several opportunities to exploit heterogeneity for increased performance or energy efficiency via a framework such as the 10x10 paradigm and architecture [25,5,1]. Here we discuss how the empirical characterization results can be coupled to the specialized micro-engines proposed in [5]. Specifically, we find plausible evidence to support four distinct customized micro-engines in the UHPC Challenge problem characterization results.

Irregular Graph - Local micro-engine

The impact of long memory latencies and the block memory structure cause irregular graph applications such as UHPC Graph to execute but poorly on traditional architectures–using memory bandwidth and the processing pipe inefficiently. The compute idioms used by the challenge problems indicate significant use of irregular memory references. Specifically, four of the challenge problems (Chess, Graph, Molecular Dynamics, and Streaming Sensor) would clearly benefit from irregular addressing support. For example, to save energy and increase efficiency a specialized graph engine would likely lower the frequency of an integer-only processor generating the DRAM references more efficiently, ordering/organizing them to match the memory system, and would provide lightweight data synchronization triggers when there are sufficient memory values for efficient computation.

Complex Task micro-engine

The distinctive characteristic of Chess is tree walking and search, pruning, semantic network. Critical bottlenecks include dynamic memory management and task management. The joules/memop figure shows it too needs but a small L2 cache and no floating-point unit. Extremely simple cores (Chess doesn't even use binw) with lightweight threads and lightweight synchronization such as have been proposed by Denneau [25] could improve efficiency and performance significantly.

Image/Media micro-engine

SAR kernel (back projection) is extremely branchy and spends significant time moving data and doing scalar single precisions floating point operations. In other dialects of these applications, applications may use short fixed precision representations values (8, 16, or 32) in packed representations, and with data-type specific operations – thresholding, scaling, shifting, etc. Special operations dealing with these datatypes, packed representations, compression, and short vector parallelism could increase efficiency significantly [26,27]. Major elements of traditional microprocessors may not be required – full floating point implementation – key operations such as small building-block FFT's might receive and multiplication by complex "twiddle factors" could receive direct hardware support.

GPU Warp micro-engine and multicores

The Shock problem is not branch intensive and is highly floating-point intensive. This is a characteristic that has been found to be indicative of suitability for GPU acceleration [28, 29] however the code makes good use of L2 cache so a traditional multicore with beefy memory subsystem may be a good option here (or increase memory per thread on GPU). The MD problem appears too branch to be suitable for GPU and appears to make good use of long cache lines and prefetching (smooth curve for larger L2 caches).

6.0 Recommendations

AACE Phase 1 clearly demonstrated the potential for architecture aware compilers to dramatically reduce application development costs and labor; ensure that executable code is optimal, correct, and timely; provide the full capabilities of computing system advances to our warfighters; and provide superior design and performance capabilities across a broad range of applications. We subjected the T1 performers to rigorous blind testing that proved their technology was on-track to meet the stated goals.

By our definition, performance idioms are the basic components of scientific applications. In this project, we enhanced automatic idioms recognition methods and implemented the method, based on the open source compiler Open64 to the UHPC "Challenge Problems." One of our next steps is to calculate the dynamic code coverage by combining the static results with the dynamic profile information. With dynamic code coverage, we then can approximate the application performance automatically and check more application codes to test our hypothesis about performance approximation. We are also working to apply the technique for performance optimization on GPU and FPGA machines. Moreover, according to our previous research on non-volatile memory, performance behaviors on these new storage technologies are totally different from traditional spinning disks. As a result, similar ideas can be applied for hybrid storage systems.

Also, future work should include characterization of the parallelization opportunities in these challenge problems. Our early results show operation and data types diversity – integer, image, binary, and floating point. They also show memory reference structure variety – regular, irregular and cacheable and non-cacheable. We know from examining the codes that there is also significant variation in control structure – tasking for search, iteration in numerical solvers in other cases. Together this diversity represents significant opportunity for customization – and more efficient heterogeneous solutions. The specific characterization results we have presented are doubtless just a beginning – we expect they will be studied in great depth by the community and PIR will be further augmented.

References

[1] Shekhar Y. Borkar and Andrew A. Chien, "The Future of Microprocessors," Communications of the Association for Computing Machinery (CACM), May 2011.

[2] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*, June 2011.

[3] Kogge, et al. ExaScale Computing Study: Technology Challenges in Achieving ExaScale Systems, DARPA ExaScale Hardware Study, 2008.

[4] Sarkar, et al. ExaScale Software Study: Software Challenges in Extreme Scale Systems, DARPA ExaScale Software study, 2009.

[5] Andrew A. Chien, Allan Snavely, and Mark Gahagan: 10x10: A General-purpose Architectural Approach to Heterogeneity and Energy Efficiency. Procedia CS 4: 1987-1996 (2011)

[6] Soumekh, M., Synthetic Aperture Radar Signal Processing with MATLAB Algorithms, Wiley Interscience, 1999.

[7] C. McCarthy, Facebook hits 500 million, aims for more growth. CNET News, July 2010, http://www.zdnet.com/news/facebook-hits-500-million-aims-for-more-growth/447195

[8] Tweets hit 50M per day, http://blog.twitter.com/2010/02/measuring-tweets.html, February 2010

[9] M. E. J. Newman, The structure and function of complex networks, SIAM Review, 45(2):167-256, 2003.

[10] D.Chakrabarti, Y.Zhan, and C.Faloutsos. R-MAT: A recursive model for graph mining. In Proc. 4th SIAM Intl. Conf. on Data Mining (SDM), Orlando, FL, April 2004. SIAM. (R-Mat)

[11] David Levy and Monty Newborn, (1991). How Computers Play Chess. Computer Science Press. ISBN 0-7167-8121-2

[12] Hsu, Feng-hsiung (2002). Behind Deep Blue: Building the Computer that Defeated the World Chess Champion. Princeton University Press. ISBN 0-691-09065-3

[13] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin: Best-First Fixed-Depth Game-Tree Search in Practice. IJCAI 1995: 273-281 (MTD-f algorithm)

[14] Albert Lindsey Zobrist, A New Hashing Method with Application for Game Playing, Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, (1969).

[15] S.Adcock and J.Andrew McCammon, Molecular Dynamics: Survey of Methods for Simulating the Activity of Proteins, Chem Rev. 2006 May; 106(5): 1589–1615. doi: 10.1021/cr040426m.
S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, J Comp Phys, 117, 1-19 (1995). http://lammps.sandia.gov

[16] ALE3D Web Site. https://wci.llnl.gov/codes/ale3d/.

[17] J.He, A.Snavely, R.Van der Wijngaart, and M.Frumkin, *Automatic Recognition of Performance Idioms in Scientific Applications*, in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, Alaska, May 16-20, 2011.

[18] MLaurenzano, M.Tikir, L.Carrington, and A.Snavely, PEBIL: Efficient Static Binary Instrumentation for Linux, Proceedings of the International Symposium for Performance Analysis of Systems and Software (ISPASS), White Plains, NY. March 2010. *PEBIL-ispass10*

[19] J. Renau, et al. The SESC Simulator, January 2005, *http://sesc.sourceforge.net*

[20] The Standard Performance Evaluation Corporation (SPEC), The SPEC Benchmarks, http://www.spec.org/benchmarks

[21] J.Dongarra and P.Luszczek, "Introduction to the HPCChallenge Benchmark Suite," ICL Technical Report, ICL-UT-05-01, (Also appears as CS Dept. Tech Report UT-CS-05-544), 2005.

[22] K.Asanovic, et al. *The Landscape of Parallel Computing Research: a View from Berkeley*, UC Berkeley EECS Technical Report UCB/EECS-2006-183, December 18, 2006.

[23] Christian Bienia, Benchmarking Modern Multiprocessors, Ph.D. Thesis. Princeton University, January 2011. (PARSEC)

[24] Andrew A. Chien, "10x10 must replace 90/10", in Proceedings of the Salishan Conference on High Performance Computing, April 2010

[25] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao: Toward a Software Infrastructure for the Cyclops-64 Cellular Architecture. HPCS 2006: 9 (Cyclops)

[26] Intel Processor MMX Documentation, http://www.intel.com/design/archives/Processors/mmx/, 1995.

[27] ARM NEON Documentation. http://www.arm.com/products/processors/technologies/neon.php

[28] Nvidia's Next Generation Compute Architecture: Fermi. Nvidia whitepaper, available from http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[29] Authors redacted, Modeling and Predicting Application Performance on Hardware Accelerators, submitted to IISWC 2011

[30] The International Technology Road Map for Semiconductors (ITRS), 2010 edition. http://www.itrs.net

[31] S.Thoziyoor, et al. CACTI 5.1 Technical Report, HPL-2008-20, Hewlett Packard Labs, April 2008.

Appendix A - x86 Opcode Classifications

Instruction Type	Data Size	Opcodes
BINARY	BYTE	seto, setno, setb, setnb, setz, setnz, setbe, seta, sets, setns, setp, setnp, setl, setge, setle, setg
	DOUBLEWORD	cwd, cwde
	QUADWORD	cdqe, cdq, cqo
	VARIABLE	shr, shld, shrd, test, xor
	WORD	Cbw
BINARYV	BYTE	pshufb, pminsb, pmaxsb
	DOUBLE	andpd, andnpd, orpd, shufpd, xorpd, unpckhpd, unpcklpd
	DOUBLEWORD	pshufd, pslld, psrad, psrld, punpckhwd, punpcklwd
	QUADWORD	pslldq, psllq, psrlq, psrldq, punpckhdq, punpckhqdq, punpckldq,punpcklqdq
	SINGLE	andps, andnps, orps, shufps, xorps, unpckhps, unpcklps
	VARIABLE	palignr, pand, pandn, por, pxor
	WORD	pshufhw, pshuflw, pshufw, psllw, psraw, psrlw, punpckhbw, punpcklbw
CACHE	N.A.	clflush, invd, invlpg, invlpga, prefetch, prefetchnta, prefetcht0, prefetcht1, prefetcht2, int, int1, int3, into, iretd, iretq, iretw, syscall, sysenter, sysexit, sysret
COND	N.A.	ja, jae, jb, jbe, jcxz, jecxz, jg, jge, jl, jle, jno, jnp, jns, jnz, jo, jp, jrcxz, js, jz
FLOAT	VARIABLE	f2xm1, fabs, fadd, faddp, fbld, fbstp, fchs, fclex, fucomi, fcomi, fucomip, fcomip, fcom, fcom2, fcomp3, fcomp, fcomp5, fcompp, fcos, fdecstp, fdiv, fdivp, fdivr, fdivrp, fiadd, fidivr, fidiv, fisub, fisubr, ficom, ficomp, fmul, fmulp, fimul, fpatan, fprem, fprem1, fptan, frndint, fscale, fsin, fsincos, fsqrt, fsub, fsubp, fsubr, fsubrp, ftst, fucom, fucomp, fucompp, fxam,

		fpxtract, fyl2x, fyl2xp1
FLOATS	DOUBLE	addsd, comisd, cvtsd2si, cvtsd2ss, cvtss2sd, cvtsi2sd, divsd, maxsd, minsd, mulsd, roundsd, sqrtsd, subsd, ucomisd
	SINGLE	addss, cmpss, comiss, cvtsi2ss, cvtss2si, cvttsd2si, cvttss2si, divss, maxss, minss, mulss, rcpss, rounds, rsqrtss, sqrtss, subs, ucomiss
FLOATV	DOUBLE	addpd, addsubpd, cmppd, cvtpd2dq, cvtpd2pi, cvttpd2pi, cvtdq2pd, cvtpi2pd, cvtps2pd, cvttpd2dq, divpd, haddpd,hsubpd, maxpd, minpd, mulpd, roundpd, sqrtpd, subpd
	SINGLE	addps, addsubps, cmpps, cvtps2dq, cvtps2pi, cvtdq2ps, cvtpd2ps, cvtpi2ps, cvttps2dq, cvttps2pi, divps, haddps, hsubps, maxps, minps,mulps, rcpps, roundps, rsqrtps, sqrtps, subps
INT	VARIABLE	adc, add, cmp, cmpxchg, cmpxchg8b, dec, div, idiv, imul, inc, mul, neg, sbb, sub, xadd, xchg
INTV	BYTE	packsswb, packuswb, paddb, paddsb, paddusb, pavgb, pcmpeqb, pcmpgtb,pmaxub, pminub
	DOUBLEWORD	pcmpeqd, pcmpgtd, pf2id, pfacc, pfadd, pfcmpeq, pfcmpge, pfcmpgt, pfmax, pfmin, pfmul, pfnacc, pfpnacc, pfrcp, pfrcpit1, pfrcpit2, pfrspit1, pfrsqrt, pfsub, pfsubr, pi2fd, paddd, packssdw, phaddd, maddwd, pminsd, pminud, pmaxsd, pmaxud, pmuludq, psubd, pswapd
	QUADWORD	paddq, psubq
	WORD	pavgw, pcmpeqw, pcmpgtw, pextrw, pinsrw, pmaxsw, pminsw, pmulhuw, pmulhw, pmullw, pavgusb, psubb, psubsb, psubusb, psadbw, psubw, psubsw, psubusw, pi2fw, pf2iw, paddw, paddsw, paddusw, pminuw, pmaxuw, pmulhrw
Invalid		d3vil, db, grp_asize, grp_mod, grp_mode, grp_osize, grp_reg, grp_rm, grp_vendor, grp_x87, invalid, na, none, ud2
MOVE	BYTE	lodsb, stosb

	DOUBLE	movhpd, movlpd, movsd
	DOUBLEWORD	lodsd, stosd, movd
	QUADWORD	lodsq, maskmovq, stosq, movntq, movq, movqa, movq2dq, movdq2q
	SINGLE	movss, movhps, movlps, movlhps, movhlps
	VARIABLE	fild, fist, fistp, fisttp, fld, fld1, fldl2t, fldl2e, fldlpi, fldlg2, fldln2, fldz, cmovo, cmovno, cmovb, cmovae, cmovz, cmovnz, cmovbe, cmova, cmovs, cmovns, cmovp, movnp,cmovl, cmovge, cmovle, cmovg, fcmovb, fcmove, fcmovbe, fcmovu, fcmovnb, fcmovne, fcmovnbe, fcmovnu, fxch, fxch4, fxch7, fstp, fstp1, fstp8, fstp9, fst, lddqu, lds, lea, les, lfs, lgs, lss, str, mov, movapd, movaps, movddup, movdqa, movdqu, movmskpd, movmskps, movntdq, movnti, movntpd, movntps, movsldup, movshdup, movsx, movupd, movups, movzx, movsxd, pmovmskb
	WORD	lodsw, ldmxcsr
OTHER		3dnow, aaa, aad, aam, aas, arpl, clc, cld, clgi, cli, clts, cmc, cupid, daa, das, emms, femms, ffree, ffreep, fldcw, fldenv, fncstp, fninit, fnop, hlt, in, insb, insd, insw, lahf, lar, lfence, lgdt, lidt, lldt, lmsw, lock, loop, loope, loopnz, lsl, ltr, mfence, monitor, mwait, nop, out, outsb, outsd, outsq, outsw, pause, rdmsr, rdpmc, rdtsc, rdtscp, rsm,sahf, sfence, sgdt, sidt, skinit, sldt, smsw, stc, std, stgi, sti, stmxcsr, swapgs, verr, verw, vmcall, vmclear, vmload, vmmcall, vmptrld, vmptrst, vmresume, vmrun, vmsave, vmxoff, vmxon, wait, wbinvd, wrmsr, xlatb
STACK		enter, leave, fnsave, fnstcw, fnstenv, fnstsw, frstor, fxrstor, fxsave, pop, push, popa, pusha, popad, pushad, popfw, pushfw, popfd, pushfd, popfq, pushfq
STRING	BYTE	movsb, cmpsb, scasb
	DOUBLEWORD	cmpsd, scads
	QUADWORD	cmpsq, movsq, scasq
	WORD	cmpsw, movsw, scasw

		repne, rep
UNCOND		call, ret, retf, jmp

Appendix B - PIR Tool Setup and Usage

B.1 Setup

Requirements Running

Running the PIR tool requires an installation of gcc >= 4.5.0.

Building

Building the PIR tool requires the plugin headers for gcc which should be found at PREFIX/lib/gcc/x86 64-unknown-linux-gnu/4.5.0/plugin/include or similar. Building may also require two additional headers which can be copied from the gcc source: tree-scalar-evolution.h and tree-chrec.h.

Installation From the binary distribution

If you already have the PIR binary, pir gcc plugin.so, there is nothing to do here.

From the source distribution

To build the tool from source, do the following:

cd /PATH/TO/pir ./configure CC=/PATH/TO/gcc make

The gcc used here must have its plugin headers installed somewhere. You can check if this location exists with:

gcc -print-file-name=plugin

If this produces no output, the plugin headers were never installed. You can either build your own gcc which should install these headers by default or you can modify GCC PLUGINS DIR in pir/rules.mk to point to a directory where you already have these headers.

Configuration

PIR comes with scripts that can be useful for testing PIR on single files, pir and pirf. If you wish to use these, you must modify configs.sh appropriately for your environment.

B.2 Invoking the Tool

PIR is implemented as a plugin to gcc and can be invoked via gcc by passing gcc (or gfortran) the argument '-fplugin=/PATH/TO/pir gcc plugin.so'.

Arguments can be passed to PIR by passing corresponding arguments to the invocation of gcc. Arguments to PIR are differentiated from other arguments in that they are prefixed by '-fplugin-arg-pir gcc plugin.so'. PIR supports the following arguments:

-idioms=(file) REQUIRED Specifies the input idiom descriptions file.

-output=(file) OPTIONAL Specifies the file that PIR should write its output to. If this argument is

omitted, PIR writes its output to stdout.

-append OPTIONAL Specifies that PIR should be append results to the
output file. By default, PIR overwrites the output file. This is useful for
running PIR on multiple source files before using the results.

In order for PIR to be run, optimization must be set to at least -O1.

For example, to invoke PIR on a Fortran source file, source.f, reading its idiom descriptions from idioms.txt and appending output to pir results, use the command:

```
gfortran -fplugin=/PATH/TO/pir_gcc_plugin.so
        -fplugin-arg-pir_gcc_plugin-output=pir_results
-fplugin-arg-pir_gcc_plugin-idioms=idioms.txt
        -fplugin-arg-pir_gcc_plugin-append
        source.f -I /PATH/TO/headers -O1
```

Arguments to PIR must be specified after the plugin itself is declared on the command line.

To use PIR to analyze source code in an existing build system, such as make, configure the build system to use gcc and/or gfortran and add the appropriate arguments from above to CFLAGS and/or FFLAGS.

B.3 Idiom Specification

The PIR tool requires an input idiom descriptions file which describes the code shapes to search for. This file is a series of idiom templates identified by the keyword idiom, followed by its name. Each idiom template has some number of for loop declarations which are used to declare induction variables, followed by a series of statement templates. For example, a stream idiom might be written as:

```
idiom stream for i, j with equal non-zero constant stride A[i] = B[j]
```

As an example of using multiple loop declarations, the transpose idiom can be written as:

```
idiom transpose for i, k with equal non-zero constant stride for j, l with equal non-zero constant stride
A[j][i] = B[k][l]
```

A for loop declaration has the form:

```
for variables with qualifiers stride
```

There must be at least one variable specified but the qualifiers list may be empty. Order is not important. The qualifiers include non-zero, constant, equal, and any. any may not be used with constant or non-zero and is the default if no qualifiers are specified.

If a variable does not appear in a for loop declaration, it is assumed to not be an induction variable and will match anything.

B.4 PIR Output

The output produced by the PIR tool is condensed. To generate a human readable file, use the script:

postprocess.pl <file>

This script formats the information into a table and is capable of retrieving source code lines to display with the idiom matches. To use code retrieval, the source code must be in a subdirectory of the current directory. The output contains several columns:

Idiom The name of the idiom that matched this statement

File The file that contains this match

Function The function that contains this match

Line The line number of the match

Depth The loop depth of the match

Start The start line of the containing loop

End The end line of the containing loop

Confidence A number between 0 and 1 representing how confident PIR is of the correctness of the match

Code The source code retrieved for the match

The output of this script is written to stdout.

Appendix C

C.1 Detailed Scores: PACE

This section enumerates the detailed scoring results for PACE on each of the public and private machines. The scoring tables use color to indicate characteristics where PACE's measured values differ from our truth values. Values colored in (red) highlight instances where there are significant differences between the measured and truth values. For each of these cases, a detailed discussion is presented in Section 10. Values colored in (yellow) highlight instances where there are only minor differences between the measured and truth values.

Several public machine truth values in these tables differ from previously published values in March 2011. These differences reflect further refinement of our truth values based on feedback from T1 groups regarding the semantics of the characteristics.

All of these values are "as given" by the PACE analysis tools without the "fix" described on page 43.

C.1.1 - Table 13: Private Machine: PS3-PPE (PACE)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	5	-1	32768	32768	0.0000	0.0000
L2 cache size	Bytes	3	5	-1	360448	393216	0.3352	0.1124
L1 TLB size	Bytes	2	3	-1	262144	262144	0.0000	0.0000
L2 TLB size	Bytes	2	3	-1	4194304	4194304	0.0000	0.0000
L1 line size	Bytes	3	5	-1	128	128	0.0000	0.0000
L2 line size	Bytes	3	5	-1	128	128	0.0000	0.0000
TLB page size	Bytes	2	5	-1	4096	4096	0.0000	0.0000
L1 associativity	Integer	3	5	-1	4	4	0.0000	0.0000
L1 latency	Ratio to Integer Add	3	5	1	2.48	2	0.9965	0.9931
L2 latency	Ratio to Integer Add	3	5	1	20	20	0.0000	0.0000
Operation Cost (+) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT32)	Ratio to Integer Add	3	3	1	5	5	0.0000	0.0000

Operation Cost (/) (INT32)	Ratio to Integer Add	3	3	1	11	16	1.4961	2.2383
Operation Cost (+) (INT64)	Ratio to Integer Add	3	3	1	2	1	1.6245	2.6390
Operation Cost (-) (INT64)	Ratio to Integer Add	3	3	1	2	1	1.6245	2.6390
Operation Cost (*) (INT64)	Ratio to Integer Add	3	3	1	21	23	0.1048	0.0110
Operation Cost (/) (INT64)	Ratio to Integer Add	3	3	1	144	200	1.1929	1.4231
Operation Cost (+) (FP32)	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
Operation Cost (-) (FP32)	Ratio to Integer Add	3	3	1	4	2	1.6245	2.6390
Operation Cost (*) (FP32)	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
Operation Cost (/) (FP32)	Ratio to Integer Add	3	3	1	32	34	0.0506	0.0026
Operation Cost (+) (FP64)	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
Operation Cost (-) (FP64)	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
Operation Cost (*) (FP64)	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
Operation Cost (/) (FP64)	Ratio to Integer Add	3	3	1	32	34	0.0506	0.0026
Simultaneous (+) (INT32)	Integer	3	5	1	2	2	0.0000	0.0000
Simultaneous (-) (INT32)	Integer	3	5	1	2	2	0.0000	0.0000
Simultaneous (*) (INT32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (/) (INT32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (-) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (*) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (/) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (FP32)	Integer	3	5	1	10	3	2.9515	8.7112
Simultaneous (-) (FP32)	Integer	3	5	1	10	2	2.9904	8.9425
Simultaneous (*) (FP32)	Integer	3	5	1	10	3	2.9515	8.7112
Simultaneous (/) (FP32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (FP64)	Integer	3	5	1	10	2	2.9904	8.9425
Simultaneous (-) (FP64)	Integer	3	5	1	10	3	2.9515	8.7112
Simultaneous (*) (FP64)	Integer	3	5	1	10	3	2.9515	8.7112

Simultaneous (/) (FP64)	Integer	3	5	1	1	1	0.0000	0.0000
32 int live values	Integer	2	5	1	32	29	0.2338	0.0547
64 int	Integer	2	5	1	16	13	0.6362	0.4047
32 float	Integer	2	5	1	32	32	0.0000	0.0000
64 float	Integer	2	5	1	32	32	0.0000	0.0000
integer contexts	Integer	2	5	1	1	1	0.0000	0.0000
float contexts	Integer	2	5	1	1	1	0.0000	0.0000
memory contexts	Integer	2	5	1	1	2	2.0000	4.0000
totals		137					32.6512	71.5646
SCORE	93.83%							
QUALITY METRIC	76.17%							

C.1.2. - Table 14: Private Machine: SDR (PACE)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	5	-1	32768	32768	0.0000	0.0000
L2 cache size	Bytes	3	5	-1	4333568	5242880	1.1107	1.2336
L1 TLB size	Bytes	2	3	-1	65536	65536	0.0000	0.0000
L2 TLB size	Bytes	2	3	-1	1048576	1048576	0.0000	0.0000
L1 line size	Bytes	3	5	-1	64	64	0.0000	0.0000
L2 line size	Bytes	3	5	-1	64	128	3.0000	9.0000
TLB page size	Bytes	2	5	-1	4096	4096	0.0000	0.0000
L1 associativity	Integer	3	5	-1	8	8	0.0000	0.0000
L1 latency	ratio to integer add	3	5	1	3	3	0.0000	0.0000
L2 latency	ratio to integer	3	5	1	15	15	0.0000	0.0000

	add							
Operation Cost (+) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (/) (INT32)	Ratio to Integer Add	3	3	1	17	17	0.0000	0.0000
Operation Cost (+) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT64)	Ratio to Integer Add	3	3	1	4.7	5	0.0524	0.0027
Operation Cost (/) (INT64)	Ratio to Integer Add	3	3	1	37.7	38	0.0025	0.0000
Operation Cost (+) (FP32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (-) (FP32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (*) (FP32)	Ratio to Integer Add	3	3	1	3.59	4	0.1445	0.0209
Operation Cost (/) (FP32)	Ratio to Integer Add	3	3	1	12.83	14	0.0971	0.0094
Operation Cost (+) (FP64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (-) (FP64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (*) (FP64)	Ratio to Integer Add	3	3	1	4.46	5	0.1603	0.0257
Operation Cost (/) (FP64)	Ratio to Integer Add	3	3	1	20.76	22	0.0469	0.0022

Simultaneous (+) (INT32)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (-) (INT32)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (*) (INT32)	Integer	3	5	1	3	3	0.0000	0.0000
Simultaneous (/) (INT32)	Integer	3	5	1	2	2	0.0000	0.0000
Simultaneous (+) (INT64)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (-) (INT64)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (*) (INT64)	Integer	3	5	1	3	2	1.9182	3.6795
Simultaneous (/) (INT64)	Integer	3	5	1	2	1	2.6330	6.9329
Simultaneous (+) (FP32)	Integer	3	5	1	3	2	1.9182	3.6795
Simultaneous (-) (FP32)	Integer	3	5	1	3	3	0.0000	0.0000
Simultaneous (*) (FP32)	Integer	3	5	1	4	3	1.3895	1.9307
Simultaneous (/) (FP32)	Integer	3	5	1	2	3	2.8428	8.0813
Simultaneous (+) (FP64)	Integer	3	5	1	3	5	2.9938	8.9627
Simultaneous (-) (FP64)	Integer	3	5	1	3	5	2.9938	8.9627
Simultaneous (*) (FP64)	Integer	3	5	1	5	3	2.2632	5.1221
Simultaneous (/) (FP64)	Integer	3	5	1	2	5	3.0000	9.0000
32 int live values	Integer	2	5	1	15	15	0.0000	0.0000

64 int	Integer	2	5	1	15	15	0.0000	0.0000
32 float	Integer	2	5	1	16	16	0.0000	0.0000
64 float	Integer	2	5	1	16	16	0.0000	0.0000
integer contexts	Integer	2	5	1	8	8	0.0000	0.0000
float contexts	Integer	2	5	1	8	8	0.0000	0.0000
memory contexts	Integer	2	5	1	8	8	0.0000	0.0000
totals		137					38.4731	102.0860
SCORE	92.62%							
QUALITY METRIC	71.92%							

C.1.3 - Table 15: Private Machine: Triton (PACE)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	5	-1	32768	32768	0.0000	0.0000
L2 cache size	Bytes	3	5	-1	262144	196608	1.2335	1.5215
L3 cache size	Bytes	1	4	-1	4194304	5242880	0.3224	0.1039
L1 TLB size	Bytes	2	3	-1	262144	262144	0.0000	0.0000
L2 TLB size	Bytes	2	3	-1	4194304		1.9375	3.7539
L1 line size	Bytes	3	5	-1	64	64	0.0000	0.0000
L2 line size	Bytes	3	5	-1	64	64	0.0000	0.0000
L3 line size	Bytes	1	4	-1	64	64	0.0000	0.0000
TLB page size	Bytes	2	5	-1	4096	4096	0.0000	0.0000
L1 associativity	Integer	3	5	-1	8	8	0.0000	0.0000
L1 latency	ratio to integer add	3	5	1	4	4	0.0000	0.0000
L2 latency	ratio to integer add	3	5	1	10	10	0.0000	0.0000
L3 latency	ratio to integer add	1	4	1	30	20	0.4832	0.2335
Operation Cost (+) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000

Operation Cost (-) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (/) (INT32)	Ratio to Integer Add	3	3	1	23	20	0.1238	0.0153
Operation Cost (+) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (/) (INT64)	Ratio to Integer Add	3	3	1	44	38	0.1365	0.0186
Operation Cost (+) (FP32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (-) (FP32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (*) (FP32)	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
Operation Cost (/) (FP32)	Ratio to Integer Add	3	3	1	15	14	0.0251	0.0006
Operation Cost (+) (FP64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (-) (FP64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (*) (FP64)	Ratio to Integer Add	3	3	1	5	5	0.0000	0.0000
Operation Cost (/) (FP64)	Ratio to Integer Add	3	3	1	22	24	0.0966	0.0093
Simultaneous (+) (INT32)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (-) (INT32)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (*) (INT32)	Integer	3	5	1	3	3	0.0000	0.0000
Simultaneous (/) (INT32)	Integer	3	5	1	2	2	0.0000	0.0000
Simultaneous (+) (INT64)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (-) (INT64)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (*) (INT64)	Integer	3	5	1	3	4	1.9782	3.9133
Simultaneous (/) (INT64)	Integer	3	5	1	2	1	2.6330	6.9329
Simultaneous (+) (FP32)	Integer	3	5	1	3	2	1.9182	3.6795
Simultaneous (-) (FP32)	Integer	3	5	1	3	3	0.0000	0.0000
Simultaneous (*) (FP32)	Integer	3	5	1	4	3	1.3895	1.9307
Simultaneous (/) (FP32)	Integer	3	5	1	1	3	3.0000	9.0000
Simultaneous (+) (FP64)	Integer	3	5	1	3	2	1.9182	3.6795
Simultaneous (-) (FP64)	Integer	3	5	1	3	2	1.9182	3.6795

Simultaneous (*) (FP64)	Integer	3	5	1	5	2	2.8475	8.1082
Simultaneous (/) (FP64)	Integer	3	5	1	1	1	0.0000	0.0000
32 int live values	Integer	2	5	1	14	15	0.0190	0.0004
64 int	Integer	2	5	1	14	15	0.0190	0.0004
32 float	Integer	2	5	1	16	16	0.0000	0.0000
64 float	Integer	2	5	1	16	16	0.0000	0.0000
integer contexts	Integer	2	5	1	8	7	0.3568	0.1273
float contexts	Integer	2	5	1	8	7	0.3568	0.1273
memory contexts	Integer	2	5	1	8	8	0.0000	0.0000
totals		140					34.6193	82.2757
SCORE	93.52%							
QUALITY METRIC	75.27%							

C.1.4 - Table 16: Public Machine: ARM (PACE)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	5	-1	16384	16384	0.0000	0.0000
L2 cache size	Bytes	3	5	-1	262144	229376	0.2439	0.0595
L1 TLB size	Bytes	2	3	-1	32768	32768	0.0000	0.0000
L2 TLB size	Bytes	2	3	-1	262144	229376	0.1131	0.0128
L1 line size	Bytes	3	5	-1	32	32	0.0000	0.0000
L2 line size	Bytes	3	5	-1	32	32	0.0000	0.0000
TLB page size	Bytes	2	5	-1	4096	4096	0.0000	0.0000
L1 associativity	Integer	3	5	-1	4	4	0.0000	0.0000
L1 latency	ratio to integer add	3	5	1	3	3	0.0000	0.0000
L2 latency	ratio to integer add	3	5	1	33	21	2.0847	4.3460
Operation Cost (+) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000

Operation Cost (*) (INT32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (/) (INT32)	Ratio to Integer Add	3	3	1	27	24	0.0864	0.0075
Operation Cost (+) (INT64)	Ratio to Integer Add	3	3	1	2	2	0.0000	0.0000
Operation Cost (-) (INT64)	Ratio to Integer Add	3	3	1	2	2	0.0000	0.0000
Operation Cost (*) (INT64)	Ratio to Integer Add	3	3	1	7	14	2.9063	8.4463
Operation Cost (/) (INT64)	Ratio to Integer Add	3	3	1	450	474	0.0388	0.0015
Operation Cost (+) (FP32)	Ratio to Integer Add	3	3	1	72	68	0.0152	0.0002
Operation Cost (-) (FP32)	Ratio to Integer Add	3	3	1	72	69	0.0061	0.0000
Operation Cost (*) (FP32)	Ratio to Integer Add	3	3	1	46	40	0.1238	0.0153
Operation Cost (/) (FP32)	Ratio to Integer Add	3	3	1	194	180	0.0308	0.0009
Operation Cost (+) (FP64)	Ratio to Integer Add	3	3	1	102	122	0.3784	0.1432
Operation Cost (-) (FP64)	Ratio to Integer Add	3	3	1	102	123	0.4124	0.1701
Operation Cost (*) (FP64)	Ratio to Integer Add	3	3	1	85	77	0.0589	0.0035
Operation Cost (/) (FP64)	Ratio to Integer Add	3	3	1	645	651	0.0030	0.0000
Simultaneous (+) (INT32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (-) (INT32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (*) (INT32)	Integer	3	5	1	3	3	0.0000	0.0000
Simultaneous (/) (INT32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (-) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (*) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (/) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (FP32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (-) (FP32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (*) (FP32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (/) (FP32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (FP64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (-) (FP64)	Integer	3	5	1	1	1	0.0000	0.0000

Simultaneous (*) (FP64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (/) (FP64)	Integer	3	5	1	1	1	0.0000	0.0000
32 int live values	Integer	2	5	1	14	14	0.0000	0.0000
64 int	Integer	2	5	1	5	5	0.0000	0.0000
32 float	Integer	2	5	1	0	0	0.0000	0.0000
64 float	Integer	2	5	1	0	0	0.0000	0.0000
integer contexts	Integer	2	5	1	1	1	0.0000	0.0000
float contexts	Integer	2	5	1	1	1	0.0000	0.0000
memory contexts	Integer	2	5	1	1	1	0.0000	0.0000
totals		137					6.5017	13.2068
SCORE	97.35%							
QUALITY METRIC	95.25%							

C.1.5 - Table 17: Public Machine: Batcave (PACE)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	5	-1	16384	16384	0.0000	0.0000
L2 cache size	Bytes	3	5	-1	262144	262144	0.0000	0.0000
L3 cache size	Bytes	1	4	-1	6291456	5242880	0.1206	0.0145
L1 TLB size	Bytes	2	3	-1	524288	1966080	2.0000	4.0000
L2 TLB size	Bytes	2	3	-1	2097152		1.9375	3.7539
L1 line size	Bytes	3	5	-1	64	64	0.0000	0.0000
L2 line size	Bytes	3	5	-1	128	128	0.0000	0.0000
L3 line size	Bytes	1	4	-1	128	128	0.0000	0.0000
TLB page size	Bytes	2	5	-1	16384	16384	0.0000	0.0000
L1 associativity	Integer	3	5	-1	2	4	3.0000	9.0000
L1 latency	ratio to integer add	3	5	1	2	2	0.0000	0.0000

L2 latency	ratio to integer add	3	5	1	6	6	0.0000	0.0000
L3 latency	ratio to integer add	1	4	1	15	11	0.3545	0.1257
Operation Cost (+) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT32)	Ratio to Integer Add	3	3	1	3	4	0.9371	0.8782
Operation Cost (/) (INT32)	Ratio to Integer Add	3	3	1	37	38	0.0132	0.0002
Operation Cost (+) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT64)	Ratio to Integer Add	3	3	1	3	4	0.9371	0.8782
Operation Cost (/) (INT64)	Ratio to Integer Add	3	3	1	44	47	0.0587	0.0034
Operation Cost (+) (FP32)	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
Operation Cost (-) (FP32)	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
Operation Cost (*) (FP32)	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
Operation Cost (/) (FP32)	Ratio to Integer Add	3	3	1	31	31	0.0000	0.0000
Operation Cost (+) (FP64)	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
Operation Cost (-) (FP64)	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
Operation Cost (*) (FP64)	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
Operation Cost (/) (FP64)	Ratio to Integer Add	3	3	1	35	36	0.0143	0.0002
Simultaneous (+) (INT32)	Integer	3	5	1	6	6	0.0000	0.0000
Simultaneous (-) (INT32)	Integer	3	5	1	6	6	0.0000	0.0000
Simultaneous (*) (INT32)	Integer	3	5	1	8	8	0.0000	0.0000
Simultaneous (/) (INT32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (INT64)	Integer	3	5	1	6	6	0.0000	0.0000
Simultaneous (-) (INT64)	Integer	3	5	1	6	6	0.0000	0.0000
Simultaneous (*) (INT64)	Integer	3	5	1	8	8	0.0000	0.0000
Simultaneous (/) (INT64)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (FP32)	Integer	3	5	1	8	8	0.0000	0.0000
Simultaneous (-) (FP32)	Integer	3	5	1	8	8	0.0000	0.0000

Simultaneous (*) (FP32)	Integer	3	5	1	8	8	0.0000	0.0000
Simultaneous (/) (FP32)	Integer	3	5	1	1	1	0.0000	0.0000
Simultaneous (+) (FP64)	Integer	3	5	1	8	8	0.0000	0.0000
Simultaneous (-) (FP64)	Integer	3	5	1	8	8	0.0000	0.0000
Simultaneous (*) (FP64)	Integer	3	5	1	8	8	0.0000	0.0000
Simultaneous (/) (FP64)	Integer	3	5	1	1	1	0.0000	0.0000
32 int live values	Integer	2	5	1	126	117	0.1571	0.0247
64 int	Integer	2	5	1	126	117	0.1571	0.0247
32 float	Integer	2	5	1	128	126	0.0209	0.0004
64 float	Integer	2	5	1	128	126	0.0209	0.0004
integer contexts	Integer	2	5	1	64	64	0.0000	0.0000
float contexts	Integer	2	5	1	64	46	1.0654	1.1351
memory contexts	Integer	2	5	1	64	64	0.0000	0.0000
totals		140					11.9521	20.5097
SCORE	96.77%							
QUALITY METRIC	91.46%							

C.1.6 - Table 18: Public Machine: DASH (PACE)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	5	-1	32768	32768	0.0000	0.0000
L2 cache size	Bytes	3	5	-1	262144	229376	0.2439	0.0595
L3 cache size	Bytes	1	4	-1	4194304	6291456	0.7606	0.5785
L1 TLB size	Bytes	2	3	-1	262144	262144	0.0000	0.0000
L2 TLB size	Bytes	2	3	-1	2097152		1.9375	3.7539
L1 line size	Bytes	3	5	-1	64	64	0.0000	0.0000
L2 line size	Bytes	3	5	-1	64	64	0.0000	0.0000

L3 line size	Bytes	1	4	-1	64	64	0.0000	0.0000
TLB page size	Bytes	2	5	-1	4096	4096	0.0000	0.0000
L1 associativity	Integer	3	5	-1	8	8	0.0000	0.0000
L1 latency	ratio to integer add	3	5	1	4	4	0.0000	0.0000
L2 latency	ratio to integer add	3	5	1	10.15	10	0.0293	0.0009
L3 latency	ratio to integer add	1	4	1	30	20	0.4832	0.2335
Operation Cost (+) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT32)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (/) (INT32)	Ratio to Integer Add	3	3	1	23	23	0.0000	0.0000
Operation Cost (+) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (-) (INT64)	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
Operation Cost (*) (INT64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (/) (INT64)	Ratio to Integer Add	3	3	1	44	41	0.0266	0.0007
Operation Cost (+) (FP32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (-) (FP32)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (*) (FP32)	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
Operation Cost (/) (FP32)	Ratio to Integer Add	3	3	1	15	14	0.0251	0.0006
Operation Cost (+) (FP64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (-) (FP64)	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
Operation Cost (*) (FP64)	Ratio to Integer Add	3	3	1	5	5	0.0000	0.0000
Operation Cost (/) (FP64)	Ratio to Integer Add	3	3	1	22	22	0.0000	0.0000
Simultaneous (+) (INT32)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (-) (INT32)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (*) (INT32)	Integer	3	5	1	3	3	0.0000	0.0000
Simultaneous (/) (INT32)	Integer	3	5	1	2	2	0.0000	0.0000
Simultaneous (+) (INT64)	Integer	3	5	1	4	1	2.9766	8.8600
Simultaneous (-) (INT64)	Integer	3	5	1	4	1	2.9766	8.8600

Simultaneous (*) (INT64)	Integer	3	5	1	3	3	0.0000	0.0000
Simultaneous (/) (INT64)	Integer	3	5	1	2	1	2.6330	6.9329
Simultaneous (+) (FP32)	Integer	3	5	1	3	2	1.9182	3.6795
Simultaneous (-) (FP32)	Integer	3	5	1	3	4	1.9782	3.9133
Simultaneous (*) (FP32)	Integer	3	5	1	4	2	2.6330	6.9329
Simultaneous (/) (FP32)	Integer	3	5	1	1	2	3.0000	9.0000
Simultaneous (+) (FP64)	Integer	3	5	1	3	2	1.9182	3.6795
Simultaneous (-) (FP64)	Integer	3	5	1	3	2	1.9182	3.6795
Simultaneous (*) (FP64)	Integer	3	5	1	5	3	2.2632	5.1221
Simultaneous (/) (FP64)	Integer	3	5	1	1	2	3.0000	9.0000
32 int live values	Integer	2	5	1	14	15	0.0190	0.0004
64 int	Integer	2	5	1	14	15	0.0190	0.0004
32 float	Integer	2	5	1	16	16	0.0000	0.0000
64 float	Integer	2	5	1	16	16	0.0000	0.0000
integer contexts	Integer	2	5	1	8	7	0.3568	0.1273
float contexts	Integer	2	5	1	8	7	0.3568	0.1273
memory contexts	Integer	2	5	1	8	7	0.3568	0.1273
totals		140					37.7831	92.3900
SCORE	93.13%							
QUALITY METRIC	73.01%							

C.2 Detailed Scores: AESOP

This section enumerates the detailed scoring results for AESOP on each of the public and private machines. The scoring tables use color to indicate characteristics where AESOP's measured values differ from our truth values. Values colored in (red) highlight instances where there are significant differences between the measured and truth values. For each of these cases, a detailed discussion is presented in Section 10 on page 37. Values colored in (yellow) highlight instances where there are only minor differences between the measured and truth values.

Several public machine truth values in these tables differ from previously published values in March 2011. These differences reflect further refinement of our truth values based on feedback from T1 groups regarding the semantics of different characteristics.

C.2.1 - Table 19: Private Machine: PS3-PPE (AESOP)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	4	-1	32768	32768	0.0000	0.0000
L2 cache size	Bytes	2	4	-1	360448	524228	1.3890	1.9293
L1 line/block size	Bytes	3	4	-1	128	128	0.0000	0.0000
L2 line/block size	Bytes	2	4	-1	128	128	0.0000	0.0000
L1 Associativity	Integer	3	4	-1	4	4	0.0000	0.0000
L2 Associativity	Integer	2	4	-1	8	4	1.5960	2.5473
32 int add latency	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	3	1	5	5.54	0.1308	0.0171
32 int div	Ratio to Integer Add	3	3	1	11	9	0.2523	0.0636
64 int add	Ratio to Integer Add	3	3	1	2	2	0.0000	0.0000
64 int sub	Ratio to Integer Add	3	3	1	2	2	0.0000	0.0000
64 int mul	Ratio to Integer Add	3	3	1	22	24	0.0966	0.0093
64 int div	Ratio to Integer Add	3	3	1	144	114	0.3339	0.1115
32 float add	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
32 float sub	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
32 float mul	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
32 float div	Ratio to Integer Add	3	3	1	32	37	0.2526	0.0638
64 float add	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
64 float sub	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
64 float mul	Ratio to Integer Add	3	3	1	4	5	0.5788	0.3351
64 float div	Ratio to Integer Add	3	3	1	32	37	0.2526	0.0638

32 int add throughput	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	5	1	0.182	0.182	0.0000	0.0000
32 int div	Ratio to Integer Add	3	5	1	0.0693	0.1103	2.9676	8.8066
64 int add	Ratio to Integer Add	3	5	1	0.678	0.663	0.0480	0.0023
64 int sub	Ratio to Integer Add	3	5	1	0.667	0.663	0.0106	0.0001
64 int mul	Ratio to Integer Add	3	5	1	0.0455	0.0414	0.3309	0.1095
64 int div	Ratio to Integer Add	3	5	1	0.00839	0.00872	0.0206	0.0004
32 float add	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	5	1	2	2.113	0.0019	0.0000
32 float div	Ratio to Integer Add	3	5	1	0.0271	0.0268	0.0209	0.0004
64 float add	Ratio to Integer Add	3	5	1	2	1.836	0.2881	0.0830
64 float sub	Ratio to Integer Add	3	5	1	2	1.836	0.2881	0.0830
64 float mul	Ratio to Integer Add	3	5	1	2.01	2.113	0.0094	0.0001
64 float div	Ratio to Integer Add	3	5	1	0.0271	0.0269	0.0133	0.0002
integer contexts	Integer	3	5	1	1	1	0.0000	0.0000
float contexts	Integer	3	5	1	1	1	0.0000	0.0000
memory contexts	Integer	3	5	1	1	1	0.0000	0.0000
NUMA node size	Integer	1	3	1	2	2	0.0000	0.0000
NUMA node count	Integer	1	3	1	1	1	0.0000	0.0000
totals		122					11.7763	15.9018
SCORE	96.73%							
QUALITY METRIC	90.35%							

C.2.2 - Table 20: Private Machine: SDR (AESOP)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	4	-1	32768	8192	2.9517	8.7125
L2 cache size	Bytes	2	4	-1	4333568	32768	1.9994	3.9977
L1 line/block size	Bytes	3	4	-1	64	1024	3.0000	9.0000
L2 line/block size	Bytes	2	4	-1	64	64	0.0000	0.0000
L1 Associativity	Integer	3	4	-1	8	8	0.0000	0.0000
L2 Associativity	Integer	2	4	-1	24	8	1.9119	3.6555
32 int add latency	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	3	1	2.9	2.7	0.0274	0.0008
32 int div	Ratio to Integer Add	3	3	1	16.8	15.3	0.0520	0.0027
64 int add	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int mul	Ratio to Integer Add	3	3	1	4.7	4.5	0.0066	0.0000
64 int div	Ratio to Integer Add	3	3	1	37.7	37.8	0.0007	0.0000
32 float add	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	3	1	3.6	3.6	0.0000	0.0000
32 float div	Ratio to Integer Add	3	3	1	12.83	12.86	0.0006	0.0000
64 float add	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	3	1	4.46	4.46	0.0000	0.0000
64 float div	Ratio to Integer Add	3	3	1	20.76	20.91	0.0022	0.0000
32 int add throughput	Ratio to Integer Add	3	5	1	2.99	2.94	0.0340	0.0012
32 int sub	Ratio to Integer Add	3	5	1	2.99	2.92	0.0515	0.0027
32 int mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 int div	Ratio to Integer Add	3	5	1	0.106	0.198	3.0000	8.9999

64 int add	Ratio to Integer Add	3	5	1	3.02	2.93	0.0703	0.0049
64 int sub	Ratio to Integer Add	3	5	1	3.02	2.93	0.0703	0.0049
64 int mul	Ratio to Integer Add	3	5	1	0.501	0.501	0.0000	0.0000
64 int div	Ratio to Integer Add	3	5	1	0.0325	0.0347	0.0197	0.0004
32 float add	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float div	Ratio to Integer Add	3	5	1	0.0834	0.0778	0.2155	0.0464
64 float add	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float div	Ratio to Integer Add	3	5	1	0.05	0.05	0.0000	0.0000
integer contexts	Integer	3	5	1	8	8	0.0000	0.0000
float contexts	Integer	3	5	1	8	8	0.0000	0.0000
memory contexts	Integer	3	5	1	8	5	2.1430	4.5926
NUMA node size	Integer	1	3	1	8	4	0.5415	0.2932
NUMA node count	Integer	1	3	1	1	2	0.9688	0.9385
totals		122					17.0671	40.2540
SCORE	94.80%							
QUALITY METRIC	86.01%							

C.2.3 - Table 21: Private Machine: Triton (AESOP)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	4	-1	32768	32768	0.0000	0.0000
L2 cache size	Bytes	2	4	-1	262144	262144	0.0000	0.0000
L1 line/block size	Bytes	3	4	-1	64	64	0.0000	0.0000
L2 line/block size	Bytes	2	4	-1	64	64	0.0000	0.0000
L1 Associativity	Integer	3	4	-1	8	8	0.0000	0.0000
L2 Associativity	Integer	2	4	-1	8	8	0.0000	0.0000
32 int add latency	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 int div	Ratio to Integer Add	3	3	1	23	23	0.0000	0.0000
64 int add	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int mul	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 int div	Ratio to Integer Add	3	3	1	44	45	0.0102	0.0001
32 float add	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	3	1	4.1	3	0.5515	0.3042
32 float div	Ratio to Integer Add	3	3	1	15	14	0.0251	0.0006
64 float add	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	3	1	5	4	0.3072	0.0944
64 float div	Ratio to Integer Add	3	3	1	22	21	0.0082	0.0001
32 int add throughput	Ratio to Integer Add	3	5	1	2.97	2.90	0.0520	0.0027
32 int sub	Ratio to Integer Add	3	5	1	2.97	2.78	0.2010	0.0404
32 int mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 int div	Ratio to Integer Add	3	5	1	0.0901	0.0926	0.0236	0.0006

64 int add	Ratio to Integer Add	3	5	1	3.02	2.95	0.0509	0.0026
64 int sub	Ratio to Integer Add	3	5	1	3.02	3.11	0.0236	0.0006
64 int mul	Ratio to Integer Add	3	5	1	1.02	1.05	0.0236	0.0006
64 int div	Ratio to Integer Add	3	5	1	0.0348	0.0365	0.0123	0.0002
32 float add	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	5	1	1.02	1.06	0.0207	0.0004
32 float div	Ratio to Integer Add	3	5	1	0.0812	0.0702	0.6018	0.3621
64 float add	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	5	1	1.02	1.06	0.0207	0.0004
64 float div	Ratio to Integer Add	3	5	1	0.0489	0.0472	0.0862	0.0074
integer contexts	Integer	3	5	1	8	8	0.0000	0.0000
float contexts	Integer	3	5	1	8	8	0.0000	0.0000
memory contexts	Integer	3	5	1	8	7	0.5352	0.2864
NUMA node size	Integer	1	3	1	4	8	0.9688	0.9385
NUMA node count	Integer	1	3	1	2	1	0.5415	0.2932
totals		122					4.0639	2.3354
SCORE	98.75%							
QUALITY METRIC	96.67%							

C.2.4 - Table 22: Public Machine: ARM (AESOP)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	4	-1	16384	16384	0.0000	0.0000
L2 cache size	Bytes	2	4	-1	262144	262144	0.0000	0.0000
L1 line/block size	Bytes	3	4	-1	32	32	0.0000	0.0000
L2 line/block size	Bytes	2	4	-1	32	32	0.0000	0.0000
L1 Associativity	Integer	3	4	-1	4	4	0.0000	0.0000
L2 Associativity	Integer	2	4	-1	4	4	0.0000	0.0000
32 int add latency	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 int div	Ratio to Integer Add	3	3	1	27	21	0.3805	0.1447
64 int add	Ratio to Integer Add	3	3	1	2	2	0.0000	0.0000
64 int sub	Ratio to Integer Add	3	3	1	2	2	0.0000	0.0000
64 int mul	Ratio to Integer Add	3	3	1	7	7	0.0000	0.0000
64 int div	Ratio to Integer Add	3	3	1	450	301	0.8184	0.6698
32 float add	Ratio to Integer Add	3	3	1	72	71	0.0015	0.0000
32 float sub	Ratio to Integer Add	3	3	1	72	71	0.0015	0.0000
32 float mul	Ratio to Integer Add	3	3	1	46	45	0.0008	0.0000
32 float div	Ratio to Integer Add	3	3	1	194	193	0.0010	0.0000
64 float add	Ratio to Integer Add	3	3	1	102	103	0.0032	0.0000
64 float sub	Ratio to Integer Add	3	3	1	102	97	0.0105	0.0001
64 float mul	Ratio to Integer Add	3	3	1	85	84	0.0015	0.0000
64 float div	Ratio to Integer Add	3	3	1	645	641	0.0011	0.0000
32 int add throughput	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 int div	Ratio to Integer Add	3	5	1	0.037	0.047	1.4219	2.0219

64 int add	Ratio to Integer Add	3	5	1	0.5	0.5	0.0000	0.0000
64 int sub	Ratio to Integer Add	3	5	1	0.5	0.5	0.0000	0.0000
64 int mul	Ratio to Integer Add	3	5	1	0.1429	0.142	0.0111	0.0001
64 int div	Ratio to Integer Add	3	5	1	0.0032	0.0033	0.0235	0.0006
32 float add	Ratio to Integer Add	3	5	1	0.0137	0.0141	0.0236	0.0006
32 float sub	Ratio to Integer Add	3	5	1	0.0137	0.0141	0.0236	0.0006
32 float mul	Ratio to Integer Add	3	5	1	0.0217	0.0222	0.0226	0.0005
32 float div	Ratio to Integer Add	3	5	1	0.0052	0.0052	0.0000	0.0000
64 float add	Ratio to Integer Add	3	5	1	0.0094	0.0096	0.0219	0.0005
64 float sub	Ratio to Integer Add	3	5	1	0.0098	0.0103	0.0097	0.0001
64 float mul	Ratio to Integer Add	3	5	1	0.0116	0.0119	0.0233	0.0005
64 float div	Ratio to Integer Add	3	5	1	0.0016	0.0016	0.0000	0.0000
integer contexts	Integer	3	5	1	1	1	0.0000	0.0000
float contexts	Integer	3	5	1	1	1	0.0000	0.0000
memory contexts	Integer	3	5	1	1	1	0.0000	0.0000
NUMA node size	Integer	1	3	1	1	1	0.0000	0.0000
NUMA node count	Integer	1	3	1	1	1	0.0000	0.0000
Totals		122					2.8014	2.8400
SCORE	98.62%							
QUALITY METRIC	97.70%							

C.2.5 - Table 23: Public Machine: Batcave (AESOP)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	4	-1	16384	16384	0.0000	0.0000
L2 cache size	Bytes	2	4	-1	262144	262144	0.0000	0.0000
L1 line/block size	Bytes	3	4	-1	64	64	0.0000	0.0000
L2 line/block size	Bytes	2	4	-1	128	128	0.0000	0.0000
L1 Associativity	Integer	3	4	-1	2	4	3.0000	9.0000
L2 Associativity	Integer	2	4	-1	8	16	2.0000	4.0000
32 int add latency	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	3	1	3	4	0.9371	0.8782
32 int div	Ratio to Integer Add	3	3	1	37	38	0.0132	0.0002
64 int add	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int mul	Ratio to Integer Add	3	3	1	3	4	0.9371	0.8782
64 int div	Ratio to Integer Add	3	3	1	44	45	0.0102	0.0001
32 float add	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
32 float div	Ratio to Integer Add	3	3	1	31	32	0.0173	0.0003
64 float add	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
64 float div	Ratio to Integer Add	3	3	1	35	36	0.0143	0.0002
32 int add throughput	Ratio to Integer Add	3	5	1	6.01	5.95	0.0186	0.0003
32 int sub	Ratio to Integer Add	3	5	1	6.01	5.95	0.0186	0.0003
32 int mul	Ratio to Integer Add	3	5	1	2.01	1.84	0.3015	0.0909
32 int div	Ratio to Integer Add	3	5	1	0.027	0.026	0.0938	0.0088

64 int add	Ratio to Integer Add	3	5	1	6.01	5.95	0.0186	0.0003
64 int sub	Ratio to Integer Add	3	5	1	6.01	5.95	0.0186	0.0003
64 int mul	Ratio to Integer Add	3	5	1	2	1.842	0.2729	0.0745
64 int div	Ratio to Integer Add	3	5	1	0.0227	0.0222	0.0477	0.0023
32 float add	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
32 float div	Ratio to Integer Add	3	5	1	0.0313	0.0357	0.3357	0.1127
64 float add	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	5	1	2	2	0.0000	0.0000
64 float div	Ratio to Integer Add	3	5	1	0.0278	0.0313	0.2489	0.0619
integer contexts	Integer	3	5	1	64	64	0.0000	0.0000
float contexts	Integer	3	5	1	64	64	0.0000	0.0000
memory contexts	Integer	3	5	1	64	4	2.9999	8.9995
NUMA node size	Integer	1	3	1	64	4	0.9922	0.9844
NUMA node count	Integer	1	3	1	1	16	1.0000	1.0000
totals		122					13.2961	26.0935
SCORE	95.81%							
QUALITY METRIC	89.10%							

C.2.6 - Table 24: Public Machine: DASH (AESOP)

Characteristic	Units	Weight	Shape (k)	Symm	Truth	Measured	Weighted Error	Weighted Error Squared
L1 cache size	Bytes	3	4	-1	32768	32768	0.0000	0.0000
L2 cache size	Bytes	2	4	-1	262144	262144	0.0000	0.0000
L1 line/block size	Bytes	3	4	-1	64	64	0.0000	0.0000
L2 line/block size	Bytes	2	4	-1	64	64	0.0000	0.0000
L1 Associativity	Integer	3	4	-1	8	8	0.0000	0.0000
L2 Associativity	Integer	2	4	-1	8	8	0.0000	0.0000
32 int add latency	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
32 int mul	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 int div	Ratio to Integer Add	3	3	1	23	23	0.0000	0.0000
64 int add	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int sub	Ratio to Integer Add	3	3	1	1	1	0.0000	0.0000
64 int mul	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 int div	Ratio to Integer Add	3	3	1	44	47	0.0587	0.0034
32 float add	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	3	1	4	4	0.0000	0.0000
32 float div	Ratio to Integer Add	3	3	1	15	14	0.0251	0.0006
64 float add	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	3	1	3	3	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	3	1	5	4.4	0.1028	0.0106
64 float div	Ratio to Integer Add	3	3	1	22	22	0.0000	0.0000
32 int add throughput	Ratio to Integer Add	3	5	1	2.96	2.79	0.1723	0.0297
32 int sub	Ratio to Integer Add	3	5	1	2.96	2.79	0.1723	0.0297
32 int mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 int div	Ratio to Integer Add	3	5	1	0.0912	0.0903	0.0184	0.0003

64 int add	Ratio to Integer Add	3	5	1	3.02	3.00	0.0118	0.0001
64 int sub	Ratio to Integer Add	3	5	1	3.02	3.00	0.0118	0.0001
64 int mul	Ratio to Integer Add	3	5	1	1.02	1.01	0.0182	0.0003
64 int div	Ratio to Integer Add	3	5	1	0.0348	0.0353	0.0175	0.0003
32 float add	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float sub	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
32 float div	Ratio to Integer Add	3	5	1	0.0714	0.0709	0.0125	0.0002
64 float add	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float sub	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float mul	Ratio to Integer Add	3	5	1	1	1	0.0000	0.0000
64 float div	Ratio to Integer Add	3	5	1	0.0455	0.0451	0.0161	0.0003
integer contexts	Integer	3	5	1	8	8	0.0000	0.0000
float contexts	Integer	3	5	1	8	7	0.5352	0.2864
memory contexts	Integer	3	5	1	8	7	0.5352	0.2864
NUMA node size	Integer	1	3	1	4	8	0.9688	0.9385
NUMA node count	Integer	1	3	1	2	1	0.5415	0.2932
totals		122					3.2179	1.8802
SCORE	98.88%							
QUALITY METRIC	97.36%							

Appendix D

D.1 Definition of Idioms

PIR includes seven idiom definitions we have found to be common in HPC applications. The idioms are described in the following section. All of the code samples are assumed to be part of a loop, i (and j) are loop induction variables.

• Stream: A[i] = A[i] + B[i]
The stream idiom includes accesses that step through arrays. In the above example two arrays are being stepped through simultaneously, but the stream idiom is not limited to this case. Stepping through any array in a loop where the index is determined by a loop induction variable is considered a stream.

• Transpose: A[i][j] = B[j][i]
The transpose idiom involves a matrix transpose, essentially reordering an array using the loop induction variable.

• Gather: A[i] = B[C[i]]
The gather idiom includes gathering data from a potentially random access area in memory to a sequential array. In this example the random accesses are created using an index array, C.

• Scatter: A[B[i]] = C[i]
The scatter idiom is essentially the opposite of gather. Values are read from a sequential area of memory and saved to an area accessed in a potentially random manner.

• Reduction: s = s + A[i]
A reduction can be formed from a stream, as in the working example, or a gather. It implies that the value returned from the read portion of the idiom is assigned to a temporary variable.

• Stencil: A[i] = A[i-1] + A[i+1]
A stencil idiom involves accessing an array in a sequential manner, including a dependency between iterations of the loop.

Bibliography

Andrew Chien, "Does 10x10 replace 90/10", Salishan'10.
http://www.lanl.gov/orgs/hpc/salishan/index10.shtml.

HPC Challenge benchmark. http://icl.cs.utk.edu/hpcc/.

Itanium Wikipedia page. http://en.wikipedia.org/wiki/Itanium.

Open Research Compiler. http://ipf-orc.sourceforge.net/.

Power4wikipediapage.http://en.wikipedia.org/wiki/POWER4.

SPEC CPU2006 home page. http://www.spec.org/cpu2006/.

STREAM: Sustainable memory bandwidth in high performance computers.
http://www.cs.virginia.edu/stream/.

Top500(Nov.2010). http://www.top500.org/lists/2010/11.

A. V. Aho, R. Sethi, and J. D. Ullman. Compilers — Principles, Techniques, and Tools. Addison Wesley, Pearson Education, Inc., 1986.

G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression using operand factorization. Proceedings of the 31th Annual International Symposium on Microarchitecture, 1998.

K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

D.Bailey,T.Harris,W.Saphir,R.vanderWijngaart,A.Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA, Dec. 1995.

K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In SC '08: Proceedings of the 2008 ACM/IEEE conference on Super-computing, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

T. Brewer. Instruction Set Innovations for Convey's HC-1 Computer. The 21st Symposium of High Performance Chips (HotChips), 2009.

A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance,

IO-Intensive Computing. In Proceedings of the2010ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of workloads used in high performance and technical computing. In ICS '07: Proceedings of the 21st annual international conference on Supercomputing, pages 73–82, New York, NY, USA, 2007. ACM.

R.Cytron, J.Ferrante, B.K.Rosen, M.N.Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4):451–490, Oct. 1991.

M. Frumkin. Data flow pattern analysis of scientific applications. In Workshop on Patterns in High Performance Computing, May 2005.

M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lain. Interprocedural parallelization analysis in SUIF. ACM Transactions on Programming Languages and Systems, 27(4):662–731, Jul. 2005.

List of Acronyms, Abbreviations and Symbols

UHPC	Ubiquitous High Performance Computing
PIR PMAC's	Idiom Recognizer
MAACE	Metrics for Architecture Aware Compilers
AACE	Architecture Aware Compilers
HPC	High Performance Computing

Do not include this….can't seem to delete it!! Pls just leave it here because it is tied to some tables in the document (I moved to footnotes under the two tables so it is covered)---- but don't include. 89 (100) is the last page. Thanks!

[i] This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}
[ii] This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}
[iii] This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}
[iv] This metric is scored for each combination of {+,-,*,/} for operation types {int32, int64, flt32, flt64}